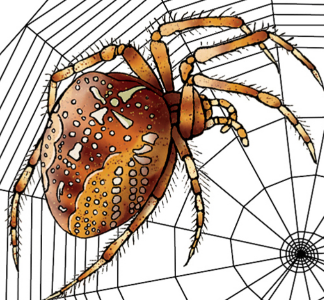
O'REILLY®

TURING 图灵程序设计丛书

数据分析之图算法

基于Spark和Neo4j

Graph Algorithms



[英] 马克·尼达姆

[美] 埃米·E. 霍德勒 著

唐富年 译



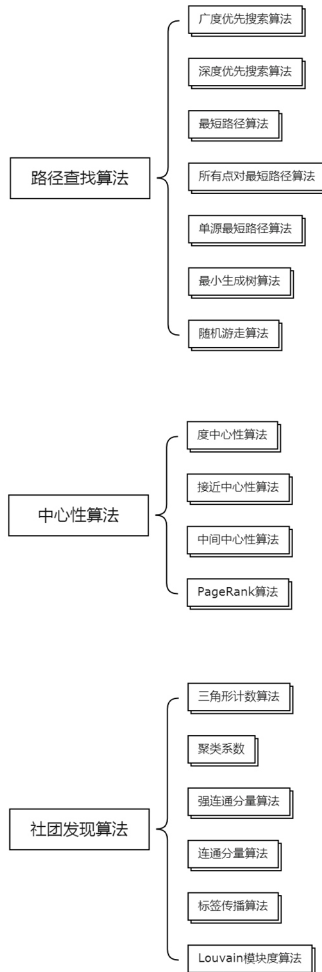
中国工信出版集团



人民邮电出版社

POSTS & TELECOM PRESS

思维导图



路径查找算法
广度优先搜索算法
深度优先搜索算法
最短路径算法
所有点对最短路径算法
单源最短路径算法
最小生成树算法
随机游走算法
中心性算法
度中心性算法
接近中心性算法
中间中心性算法
PageRank算法
社团发现算法
三角形计数算法
聚类系数
强连通分量算法
连通分量算法
标签传播算法
Louvain模块度算法

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。


图灵程序设计丛书

数据分析之图算法：基于Spark和Neo4j

Graph Algorithms

[英] 马克·尼达姆 [美] 埃米·E. 霍德勒 著
唐富年 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

数据分析之图算法 : 基于Spark和Neo4j / (英) 马克·尼达姆 (Mark Needham) , (美) 埃米·E.霍德勒 (Amy E. Hodler) 著 ; 唐富年译. -- 北京 : 人民邮电出版社, 2020.9
（图灵程序设计丛书）
ISBN 978-7-115-54667-8

Ⅰ. ①数… Ⅱ. ①马… ②埃… ③唐… Ⅲ. ①数据处理 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2020)第153025号

内 容 提 要

图分析可以揭示复杂系统和大规模网络的运作机制，图算法为构建智能应用程序提供了快速建模的框架，有助于更准确、更快速地做出预测。包括商品推荐和欺诈检测在内的许多人工智能问题能够转换为图论问题。本书基于 Spark 和 Neo4j 讲解近 20 种常用的图算法，帮助读者拓展重要图分析类型的相关知识和能力，更快速地发现数据中的模式并找到更优的解决方案。

本书适合数据分析人员、数据科学从业者，以及其他有兴趣实践图算法的读者阅读。

◆ 著　　[英] 马克·尼达姆　[美] 埃米·E. 霍德勒
　译　　唐富年
　责任编辑　谢婷婷
　责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　https://www.ptpress.com.cn
　北京　　　　　印刷

◆ 开本：800×1000　1/16
　印张：13.25
　字数：313千字　　　　2020年 9 月第 1 版
　印数：1－2 500册　　　2020年 9 月北京第 1 次印刷

著作权合同登记号　图字：01-2020-0423号

定价：79.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东市监广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly 以“分享创新知识、改变世界”为己任。40 多年来我们一直向企业、个人提供成功所必需之技能及思想，激励他们创新并做得更好。

O’Reilly 业务的核心是独特的专家及创新者网络，众多专家及创新者通过我们分享知识。我们的在线学习（Online Learning）平台提供独家的直播培训、图书及视频，使客户更容易获取业务成功所需的专业知识。几十年来 O’Reilly 图书一直被视为学习开创未来之技术的权威资料。我们每年举办的诸多会议是活跃的技术聚会场所，来自各领域的专业人士在此建立联系，讨论最佳实践并发现可能影响技术行业未来的新趋势。

我们的客户渴望做出推动世界前进的创新之举，我们希望能助他们一臂之力。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的领域，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，那就走小路。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

序 xi

前言 xiii

第 1 章　导论 1

1.1　何谓图 1
1.2　何谓图分析和图算法 3
1.3　图处理、图数据库、图查询和图算法 5
1.4　为何要关心图算法 6
1.5　图分析用例 9
1.6　小结 10

第 2 章　图论及其概念 11

2.1　术语 11
2.2　图的类型和结构 12
2.3　图的种类 14
2.3.1　连通图与非连通图 14
2.3.2　无权图与加权图 15
2.3.3　无向图与有向图 16
2.3.4　无环图与有环图 17
2.3.5　稀疏图与稠密图 18
2.3.6　单部图、二部图和 k 部图 19
2.4　图算法的类型 21
2.4.1　路径查找 21
2.4.2　中心性 21

2.4.3 社团发现 ······ 22
2.5 小结 ······ 22

第 3 章 图平台和图处理 ······ 23
3.1 图平台和图处理的注意事项 ······ 23
3.1.1 平台注意事项 ······ 23
3.1.2 处理注意事项 ······ 24
3.2 典型平台 ······ 25
3.2.1 选择平台 ······ 25
3.2.2 Apache Spark ······ 26
3.2.3 Neo4j 图平台 ······ 28
3.3 小结 ······ 30

第 4 章 路径查找算法和图搜索算法 ······ 31
4.1 示例数据：交通图 ······ 33
4.1.1 将数据导入 Spark ······ 35
4.1.2 将数据导入 Neo4j ······ 36
4.2 广度优先搜索 ······ 36
4.3 深度优先搜索 ······ 38
4.4 最短路径算法 ······ 40
4.4.1 何时使用最短路径算法 ······ 41
4.4.2 使用 Neo4j 实现最短路径算法 ······ 41
4.4.3 使用 Neo4j 实现加权最短路径算法 ······ 43
4.4.4 使用 Spark 实现加权最短路径算法 ······ 44
4.4.5 最短路径算法的变体：A* 算法 ······ 46
4.4.6 最短路径算法的变体：Yen 的 *k* 最短路径算法 ······ 48
4.5 所有点对最短路径算法 ······ 49
4.5.1 近观所有点对最短路径算法 ······ 50
4.5.2 何时使用所有点对最短路径算法 ······ 51
4.5.3 使用 Spark 实现所有点对最短路径算法 ······ 51
4.5.4 使用 Neo4j 实现所有点对最短路径算法 ······ 52
4.6 单源最短路径算法 ······ 53
4.6.1 何时使用单源最短路径算法 ······ 54
4.6.2 使用 Spark 实现单源最短路径算法 ······ 55
4.6.3 使用 Neo4j 实现单源最短路径算法 ······ 57
4.7 最小生成树算法 ······ 57
4.7.1 何时使用最小生成树算法 ······ 58
4.7.2 使用 Neo4j 实现最小生成树算法 ······ 59

4.8 随机游走算法 61
4.8.1 何时使用随机游走算法 61
4.8.2 使用 Neo4j 实现随机游走算法 61
4.9 小结 63
第 5 章 中心性算法 64
5.1 示例数据：社交图 66
5.1.1 将数据导入 Spark 67
5.1.2 将数据导入 Neo4j 67
5.2 度中心性算法 68
5.2.1 可达性 68
5.2.2 何时使用度中心性算法 69
5.2.3 使用 Spark 实现度中心性算法 69
5.3 接近中心性算法 70
5.3.1 何时使用接近中心性算法 71
5.3.2 使用 Spark 实现接近中心性算法 72
5.3.3 使用 Neo4j 实现接近中心性算法 74
5.3.4 接近中心性算法变体：Wasserman & Faust 算法 75
5.3.5 接近中心性算法变体：调和中心性算法 77
5.4 中间中心性算法 78
5.4.1 桥与控制点 78
5.4.2 计算中间中心性得分 79
5.4.3 何时使用中间中心性算法 79
5.4.4 使用 Neo4j 实现中间中心性算法 80
5.4.5 中间中心性算法变体：RA-Brandes 算法 82
5.5 PageRank 算法 83
5.5.1 影响力 84
5.5.2 PageRank 算法公式 84
5.5.3 迭代、随机冲浪者和等级沉没 85
5.5.4 何时使用 PageRank 算法 86
5.5.5 使用 Spark 实现 PageRank 算法 87
5.5.6 使用 Neo4j 实现 PageRank 算法 88
5.5.7 PageRank 算法变体：个性化 PageRank 算法 90
5.6 小结 91
第 6 章 社团发现算法 92
6.1 示例数据：软件依赖图 94
6.1.1 将数据导入 Spark 96

6.1.2 将数据导入 Neo4j ········· 97
6.2 三角形计数和聚类系数 ········· 97
6.2.1 局部聚类系数 ········· 97
6.2.2 全局聚类系数 ········· 98
6.2.3 何时使用三角形计数和聚类系数 ········· 98
6.2.4 使用 Spark 实现三角形计数算法 ········· 99
6.2.5 使用 Neo4j 实现三角形计数算法 ········· 99
6.2.6 使用 Neo4j 计算局部聚类系数 ········· 100
6.3 强连通分量算法 ········· 101
6.3.1 何时使用强连通分量算法 ········· 102
6.3.2 使用 Spark 实现强连通分量算法 ········· 102
6.3.3 使用 Neo4j 实现强连通分量算法 ········· 103
6.4 连通分量算法 ········· 106
6.4.1 何时使用连通分量算法 ········· 106
6.4.2 使用 Spark 实现连通分量算法 ········· 106
6.4.3 使用 Neo4j 实现连通分量算法 ········· 107
6.5 标签传播算法 ········· 108
6.5.1 半监督学习和种子标签 ········· 110
6.5.2 何时使用标签传播算法 ········· 110
6.5.3 使用 Spark 实现标签传播算法 ········· 110
6.5.4 使用 Neo4j 实现标签传播算法 ········· 111
6.6 Louvain 模块度算法 ········· 113
6.6.1 通过模块度进行基于质量的分组 ········· 114
6.6.2 何时使用 Louvain 模块度算法 ········· 117
6.6.3 使用 Neo4j 实现 Louvain 模块度算法 ········· 118
6.7 验证社团 ········· 122
6.8 小结 ········· 122

第 7 章 图算法实战 ········· 123

7.1 使用 Neo4j 分析 Yelp 数据 ········· 123
7.1.1 Yelp 社交网络 ········· 124
7.1.2 导入数据 ········· 124
7.1.3 图模型 ········· 125
7.1.4 Yelp 数据概览 ········· 125
7.1.5 行程规划应用程序 ········· 129
7.1.6 旅游商务咨询 ········· 134
7.1.7 查找相似类别 ········· 138

7.2 使用 Spark 分析航班数据 ······ 142
7.2.1 探索性分析 ······ 144
7.2.2 热门机场 ······ 144
7.2.3 源自 ORD 的延误 ······ 145
7.2.4 SFO 的糟糕一天 ······ 147
7.2.5 通过航空公司互连的机场 ······ 149
7.3 小结 ······ 154
第 8 章 使用图算法增强机器学习 ······ 155
8.1 机器学习和上下文的重要性 ······ 155
8.2 关联特征提取与特征选择 ······ 157
8.2.1 图特征 ······ 158
8.2.2 图算法特征 ······ 158
8.3 图与机器学习实践：链接预测 ······ 160
8.3.1 工具和数据 ······ 161
8.3.2 将数据导入 Neo4j ······ 162
8.3.3 合著者关系图 ······ 163
8.3.4 创建均衡的训练数据集和测试数据集 ······ 164
8.3.5 如何预测缺失链接 ······ 169
8.3.6 创建机器学习管道 ······ 170
8.3.7 预测链接：基本图特征 ······ 171
8.3.8 预测链接：三角形和聚类系数 ······ 181
8.3.9 预测链接：社团发现 ······ 184
8.4 小结 ······ 190
8.5 总结 ······ 190
附录 额外信息及资料 ······ 191
关于作者 ······ 195
关于封面 ······ 195

序

看看以下事项有何共同点：营销归因分析、反洗钱分析、客户旅程建模、安全事故原因分析、基于文献的发现、欺诈网络检测、互联网搜索节点分析、地图应用创建、疾病聚类分析，以及莎士比亚戏剧的剧情分析。你可能已经猜到了，上述事项的共同点是都会用到图，这证明莎士比亚所说的“全世界是一张图”[1]是正确的。

当然，莎士比亚的那句话并没有提到**图**，他写的是**舞台**。但请注意，上述例子都涉及实体与实体之间的关系，包括直接关系和间接（传递）关系。实体就是图中的节点，可以是人、事件、对象、概念或位置。节点之间的关系就是图中的边。莎士比亚戏剧的精髓不正是对实体（节点）及其关系（边）的逼真刻画吗？也许，莎士比亚真的应该在他的那句名言中写上**图**。

图算法和图数据库有趣且强大，这并不是因为两个实体之间的简单关系，即 A 与 B 相关。毕竟，数据库标准关系模型早在几十年前就在实体关系图中实例化了这些关系类型。真正使图重要的是方向关系和传递关系。在方向关系中，A 可能导致 B，反之则不一定。在传递关系中，A 可以与 B 直接相关，B 可以与 C 直接相关，而 A 与 C 不直接相关，即 A 通过传递关系与 C 相关。

利用这些传递关系，图模型能够揭示实体之间的关系，特别是当关系的数量众多且多样化时，实体之间可能存在许多关系（或网络模式）和分离度。如果没有图模型，那么实体可能看起来是不相连或不相关的，也就无法在关系数据库中表示。因此，图模型可以有效地应用于许多网络分析场景。

思考如下营销归因用例：人物 A 看到了某产品的促销活动信息，并在社交媒体上谈论该促销活动；人物 B 与人物 A 有联系，B 看到了 A 的评论，然后购买了该产品。从营销主管的视角来看，标准的关系模型无法识别这样的归因，因为 B 没有看到活动信息，而 A 又没

注 1：原话是 All the world’s a stage（全世界是一个舞台）。——译者注

有对活动做出实际响应。这样的活动看起来是失败的，但是通过图分析算法就会发现，它实际上是成功的（投资回报率为正），基于促销活动和最终客户购买之间的传递关系，通过中间人（位于中间的实体）实现。

接下来，思考一个反洗钱分析用例：人物 A 和人物 C 涉嫌非法贩运。两人之间的任何互动（例如金融数据库中的交易信息）都会被当局标记出来，并受到严格审查。然而，如果 A 和 C 从来没有过业务往来，而是通过安全、受人尊敬、未经标记的金融机构 B 进行金融交易，那么如何发现非法交易呢？用图分析算法！图引擎将发现 A 和 C 之间通过中间机构 B 的传递关系。

在互联网搜索中，主流搜索引擎使用基于图的超链接网络算法查找任意给定的检索词集合，在整个互联网上寻找中心权威节点。在这种情况下，边的方向性至关重要，因为网络中的权威节点是其他许多节点会指向的节点。

基于文献的发现（literature-based discovery，LBD）是一种基于图的知识网络应用，支持深入挖掘含有成千上万篇甚至数百万篇期刊文章的知识库。"隐性知识"只能通过已发表的研究成果之间的关系来发现，而这些研究成果之间可能存在多度分离（传递关系）。LBD 已被应用于癌症研究，该领域的医疗知识库富含语义，包含症状、诊断结果、治疗方案、药物相互作用、遗传标记、短期效果和长期作用等信息，而前所未知的治疗方法和对疑难杂症的有效治疗方案就可能"隐藏"其中。知识可能已经存在于网络之中，但是我们需要融会贯通才能发现知识。

对于前面提到的其他用例，也可以给出类似的图功能描述。每个用例都涉及实体（人、对象、事件、动作、概念和位置）及其关系（接触点，包括因果关系和简单关联）。

在考虑图的强大功能时应该牢记，对于实际用例来说，图模型中最强大的当属**上下文**。上下文包括时间、位置、相关事件、邻近实体等。将上下文作为节点和边整合到图中，可以产生惊人的预测分析和规范分析功能。

这本书旨在帮我们拓展重要图分析类型的相关知识和能力，包括算法、概念以及算法的实际机器学习应用等。从基本概念到基本算法，再到处理平台和实际用例，作者编写了一本内容翔实且富有指导性的指南，带你领略图的奇妙世界。

——Kirk Borne 博士，博思艾伦咨询公司首席数据科学家兼执行顾问，2019 年 3 月

前言

从金融系统和通信系统，到社会过程和生物过程，整个世界都是由联系驱动的。揭示这些联系背后的含义将推动各行各业在很多领域取得突破，例如识别诈骗团伙、优化建议以评估群组实力、预测级联故障等。

随着连接速度不断加快，人们对图算法的兴趣呈爆炸式增长，这不足为奇，因为图算法基于数学，其发展是为了洞悉数据间的关系。图分析可以揭示复杂系统和大规模网络的运作机制，这适用于任何组织。

图分析的实用性和重要性令人着迷，揭示复杂场景内部运作机制的过程充满乐趣。直到最近，采用图分析还需要大量专业知识和判断，因为工具的使用和集成非常困难，很少有人知道如何应用图算法来解除困惑。本书旨在改变这种现状，帮助组织更好地利用图分析，从中收获新发现，进而更快地研究出智能解决方案。

本书内容

对拥有 Apache Spark ™或 Neo4j 使用经验的开发人员和数据科学家来说，本书是帮助他们开始学习图算法的实用指南。书中的算法示例采用 Spark 和 Neo4j 两种平台来演示，但无论选择哪种图平台，本书都有助于理解常用的图概念。

前两章介绍图分析、图算法和图论。第 3 章简要介绍本书要使用的平台，第 4 ～ 6 章深入讨论经典的图算法，包括路径查找算法、中心性算法、社团发现算法等。最后两章介绍如何在工作流程中使用图算法：第 7 章介绍常规分析，第 8 章讲解机器学习。

在介绍每种算法之前，本书都会给出一份速查表，可快速跳转到相应算法。对每种算法的介绍都包含以下内容：

- 算法功能；

- 算法的用例和参考资料，以便了解更多内容；
- 示例代码，演示在 Spark、Neo4j（或两者都有）上应用算法的具体方法。

排版约定

本书使用以下排版约定。

- **黑体**

  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）

  表示程序以及段落内引用的程序元素，如变量、函数、数据库、数据类型、环境变量、语句、关键字。

- 等宽粗体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该被替换为由用户提供的值或由上下文确定的值的文本。



该图标表示提示或建议。



该图标表示普通的注记。



该图标表示警告或警示。

使用示例代码

本书的附加资料（代码示例、练习等）可以从 O'Reilly 网站[1]下载。

注 1：也可以访问本书在图灵社区的页面并下载示例代码：http://ituring.cn/book/2694。——编者注

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在自己的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN，例如 *Graph Algorithms* by Mark Needham and Amy E. Hodler (O'Reilly). Copyright 2019 Mark Needham and Amy E. Hodler, 978-1-492-04768-1。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

O'Reilly在线学习平台（O'Reilly Online Learning）

O'REILLY®

近 40 年来，O'Reilly Media 致力于提供技术和商业培训、知识和卓越见解，来帮助众多公司取得成功。

我们拥有独一无二的专家和革新者组成的庞大网络，他们通过图书、文章、会议和我们的在线学习平台分享他们的知识和经验。O'Reilly 的在线学习平台允许你按需访问现场培训课程、深入的学习路径、交互式编程环境，以及 O'Reilly 和 200 多家其他出版商提供的大量文本和视频资源。有关的更多信息，请访问 https://oreilly.com。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论和技术性问题，请发送电子邮件到 bookquestions@oreilly.com。

O'Reilly 的每一本书都有专属网页，你可以在那儿找到书的相关信息，包括勘误表[2]、示例

注 2：本书中文版勘误请到 http://ituring.cn/book/2694 查看和提交。——编者注

代码以及其他信息。本书的网页地址是 https://oreil.ly/graph-algorithms。

要了解 O’Reilly 图书、培训课程、会议和新闻的更多信息，请访问以下网站：https://www.oreilly.com。

我们在 Facebook 的地址如下：https://facebook.com/oreilly。

请关注我们的 Twitter 动态：https://twitter.com/oreillymedia。

我们的 YouTube 视频地址如下：https://www.youtube.com/oreillymedia。

致谢

我们非常享受写作本书的过程，在此感谢所有为本书撰写提供过帮助的人。特别感谢 Michael Hunger 的指导，也感谢 Jim Webber 细致的编辑和 Tomaz Bratanic 热忱的研究。最后，非常感谢 Yelp 允许我们使用其丰富的数据集，助力构建生动的示例。

电子书

扫描如下二维码，即可购买本书中文电子版。



第 1 章

导论

> 图是计算机科学的一大主题，可用于抽象表示交通运输系统、人际交往网络和电信网络。对于训练有素的程序员而言，能够用一种形式来对不同的结构建模是强大的力量之源。
>
> ——《算法设计指南》
>
> Steven S. Skiena，石溪大学计算机科学杰出授课教授

当今紧迫的数据挑战在于厘清关系，而不仅仅是对离散数据制表。图技术及其分析方法为用于研究、社会活动和商业解决方案的关联数据提供了强大的工具，举例如下：

- 对金融市场、信息技术服务等多种动态环境建模；
- 预测传染病的传播以及由此引发的服务延迟和中断；
- 查找用于机器学习的预测性特征，以打击金融犯罪；
- 发现针对个性化体验和推荐的模式。

随着数据之间的关联度日益增强，系统也越来越复杂，利用数据中丰富且不断演进的关系变得非常重要。

本章将介绍图分析和图算法。在介绍图算法和解释图数据库与图处理之间的区别之前，首先简要回顾图的起源。本章将探究现代数据的本质，揭示为何联系所含的信息要比用基本统计方法发现的信息复杂得多，最后还会研究一些图分析用例。

1.1 何谓图

图的历史可以追溯到 1736 年，即欧拉解决“哥尼斯堡七桥”问题的那一年。“哥尼斯堡七

桥”问题是指，能否参观哥尼斯堡市里由 7 座桥连接的 4 个区域，而且每座桥只允许经过一次。实际上这是不可能做到的。

欧拉意识到“哥尼斯堡七桥”问题仅与连接关系本身相关，他为图论及其数学运算奠定了基础。图 1-1 描绘了欧拉的解题过程，其中有他绘制的一张草图，摘自其论文“Solutio problematis ad geometriam situs pertinentis”[1]。

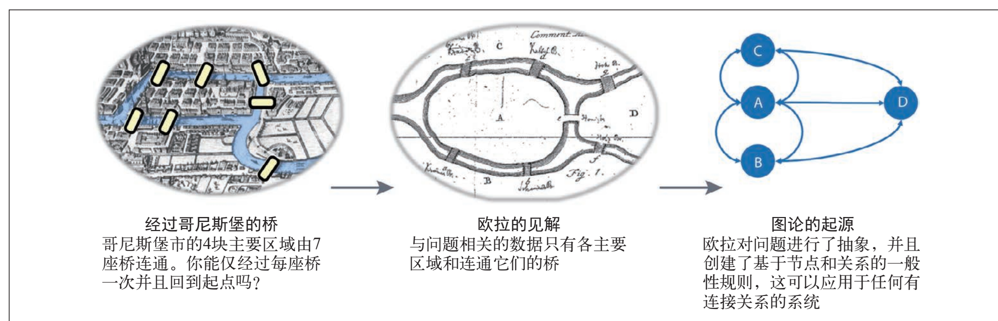


图 1-1：图论的起源。哥尼斯堡市有两座岛，它们通过 7 座桥与该市的另外两块陆地相连。难题在于如何构建一条遍历整个城市的路径，使得每座桥只经过一次

图源于数学，也是一种实用且高精度的数据建模和分析方法。构成图的对象称为**节点**或**顶点**，它们之间的关联称为**关系**、**联系**或**边**。本书使用**节点**和**关系**这两个术语。可以把节点看作句子中的名词，把关系看作为节点提供上下文的动词。为避免混淆，本书所指的图与图 1-2 中的方程式绘图或图表无关。

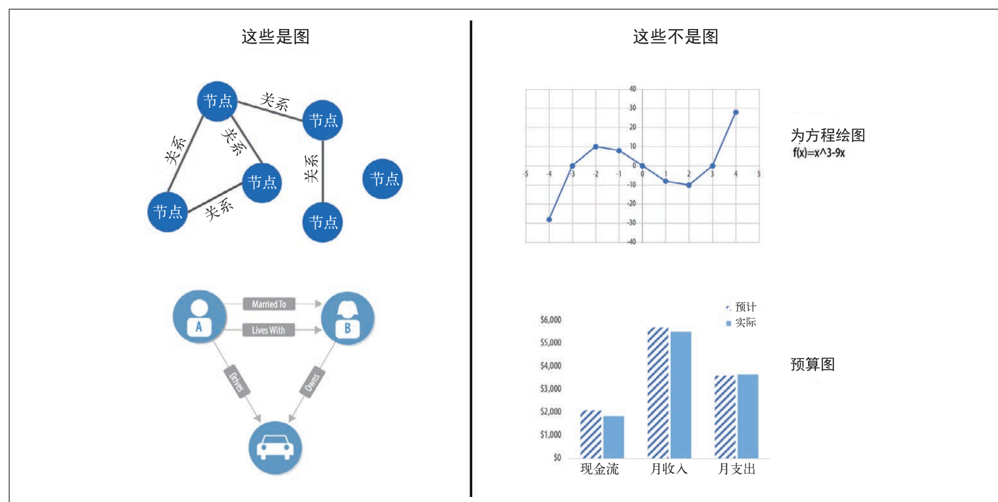


图 1-2：图是网络的一种表示方法，通常用圆表示被称为节点的实体，用线条表示关系

注 1：大意是“位置几何相关问题的解决方法”。——编者注

请看图 1-2 中的人物图，很容易就能想出几个描述该图的句子，例如人物 A 和拥有一辆车的人物 B 住在一起，人物 A 驾驶的是人物 B 的车。这种建模方法很有趣，因为易于将其与现实世界联系起来，也便于在白板上展示。这种方法有助于进行恰当的数据建模和分析。

完成图的建模才行至半途，还需要处理图，揭示其中并不显而易见的信息，而这正是图算法的用武之地。

1.2　何谓图分析和图算法

图算法属于图分析工具。图分析是使用基于图的方法来分析关联数据的过程。有多种方法可用，包括查询图数据、使用基本的统计方法、直观地研究图，或者将图整合到机器学习任务中，等等。基于图模式的查询通常用于局部数据分析，而图计算算法通常用于全局分析和迭代分析。尽管这些类型的分析方法在运用上相互交叉，但本书仍使用**图算法**这个术语来指代后者，它更多地用于计算分析和数据科学。

网络科学

网络科学是一个植根于图论的学术领域，主要研究有关对象关系的数学模型。网络科学家依赖图算法和数据库管理系统来研究数据的规模、关联性和复杂性。

在复杂性和网络科学方面有许多优质资源，下面列出一些供参考。

- Albert-László Barabási 撰写的 *Network Science*，这是一本入门电子书。
- Complexity Explorer 提供的在线课程。
- 新英格兰复杂系统研究所（New England Complex Systems Institute）提供的各种资源和论文。

图算法是分析关联数据的一种有效方法，因为图的数学运算是针对关系运算设计的。图算法描述了处理图以发现其一般性质或特定量所需的步骤。基于图论的数学原理，图算法利用节点之间的关系来推断复杂系统的组织形态和动态性。网络科学家利用这些算法来发现隐藏信息，检验假设，预测行为。

从防欺诈、优化呼叫路由到预测流感传播，图算法的应用非常广泛，例如对电力系统中达到过载条件的特定节点进行评分，或者在传输系统对应的图中发现拥塞分组。

事实上，美国民航系统在 2010 年经历了两起严重的旅客滞留事件，波及多个机场，事后人们使用图分析方法研究了这两起事件。网络科学家 P. Fleurquin、J. J. Ramasco 和 V. M. Eguíluz 使用图算法证实了这两起事件的部分原因是系统级联延迟，并在纠正建议中提到了这些信息，详见论文“Systemic Delay Propagation in the US Airport Network”。

为了将航空交通网络可视化，Martin Grandjean 在其文章“Connected World: Untangling the

Air Traffic Network”中创建了如图 1-3 所示的图。这张图清晰地显示了航空交通簇这样高度连接的结构。许多交通系统的连接呈集中式分布，明显含有可影响延迟的**中心辐射模式**（hub-and-spoke pattern）。

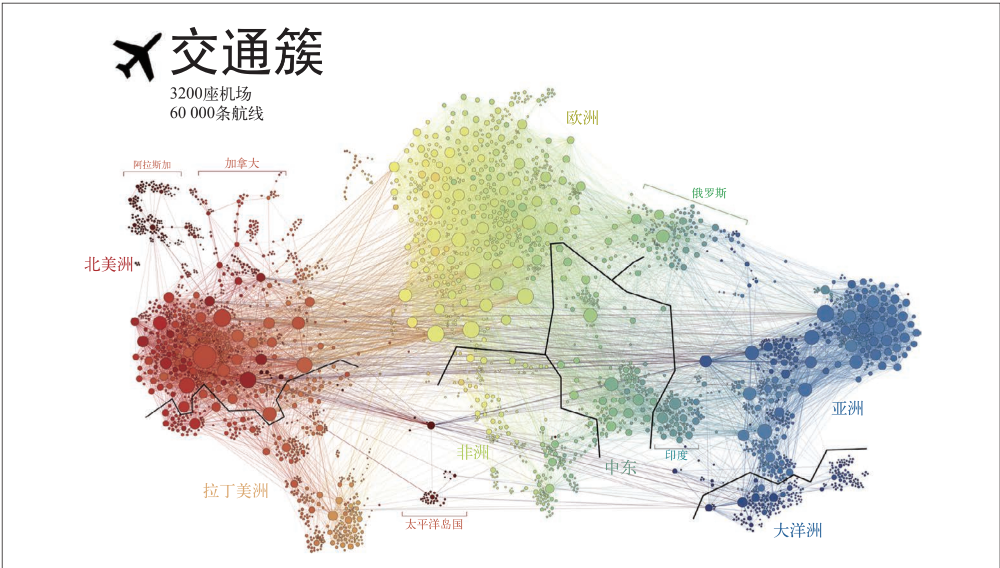


图 1-3：航空交通网络展示了在多个尺度上演进的中心辐射结构，这些结构有助于调节交通流量

图还有助于揭示那些非常微小的交互和动态变化如何引发全局突变。通过精确表示哪些事物在全局结构中存在交互，图将微观尺度和宏观尺度紧密地联系在一起。这些关联关系可用于预测行为和确定缺失的联系。图 1-4 是草原物种相互作用的食物网，可使用图分析评估层级组织和物种相互作用，然后预测缺失的关系，参见 A. Clauset、C. Moore 和 M. E. J. Newman 的论文“Hierarchical Structure and the Prediction of Missing Links in Networks”。

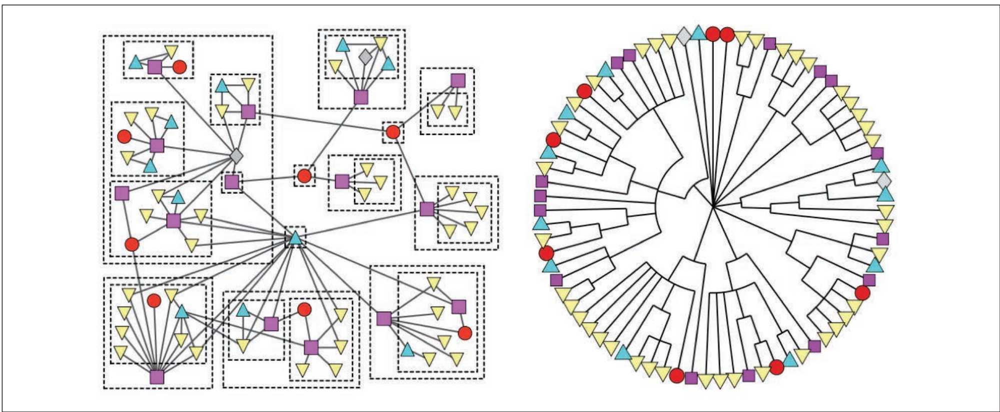


图 1-4：这张草原物种的食物网通过图将小尺度的相互作用与更大尺度的结构形态关联了起来

1.3 图处理、图数据库、图查询和图算法

图处理是指执行图工作载荷和任务的方法。大多数图查询针对图的特定部分（例如起始节点），而且要做的工作通常集中在起始节点周边的子图中。我们将这类工作称为**局部图处理**，而且这意味着要以声明方式查询图的结构，参见 Ian Robinson、Jim Webber 和 Emil Eifrem 在《图数据库》一书中的解释。这类局部图处理通常用于实时事务处理和基于模式的查询。

当谈到图算法时，经常要查找全局模式和结构。算法的输入通常是整张图，其输出则既可以是一张增强的图，也可以是某种总值，比如得分。我们将这样的处理归类为**全局图处理**，这意味着要使用计算方法（通常是迭代的）来处理图的结构。这种方法通过网络的连接关系揭示其整体性质。很多组织倾向于使用图算法来对系统建模，并且基于事物传播途径、其重要组件、群组标识和系统的整体稳健性来预测行为。

这些定义可能存在一些重叠——有时也可以通过处理算法来应答局部查询，反之亦然。但是简而言之，对整张图的操作一般通过计算方法来处理，而对子图的操作一般通过数据库来查询。

传统上，事务处理和事物分析是相互独立的，这是一种基于技术限制的非自然分离。我们认为，图分析可以驱动更智能的事务，从而为进一步分析创造了新的数据和机会。最近出现了一种趋势，那就是将这两个部分集成起来，以实现实时决策。

OLTP和OLAP

联机事务处理（online transaction processing，OLTP）通常涉及一些短期活动，比如预订机票、记入账户、预售商品等。OLTP 提供海量低延迟查询处理和高数据完整性。虽然 OLTP 的每个事务可能只涉及少量记录，但是系统要同时处理许多事务。

联机分析处理（online analytical processing，OLAP）有助于对历史数据进行更复杂的查询和分析。这些分析可能涉及多个数据源、多种格式和类型。检测趋势、执行假设场景、进行预测和发现结构模式等都是典型的 OLAP 用例。与 OLTP 相比，OLAP 系统处理的事务更少，但是处理的记录更多且运行时间更长。OLAP 系统倾向于更快地读取数据，并不希望像 OLTP 那样出现事务更新，而且通常采用面向批处理的操作。

然而近年来，OLTP 和 OLAP 之间的界限开始变得模糊。现代数据密集型应用已将实时事务操作与分析结合起来。这种处理方式上的整合是由软件领域的进步推动的，例如可伸缩性更强的事务管理和增量流处理，以及低成本、大内存硬件的支持。

将分析和事务结合起来，就可以将连续分析作为常规作业的固有组成部分。针对从 POS 机、制造系统或物联网设备收集而来的数据，当前的分析手段已经能够在处理过程中实时推荐和进行决策。这种趋势在几年前就已经出现，用于描述这种融合的术语包括**事务**

型分析（translytics）和**混合事务与分析处理**（hybrid transactional and analytical processing，HTAP）。图 1-5 演示了如何使用只读副本将不同类型的处理组合到一起。

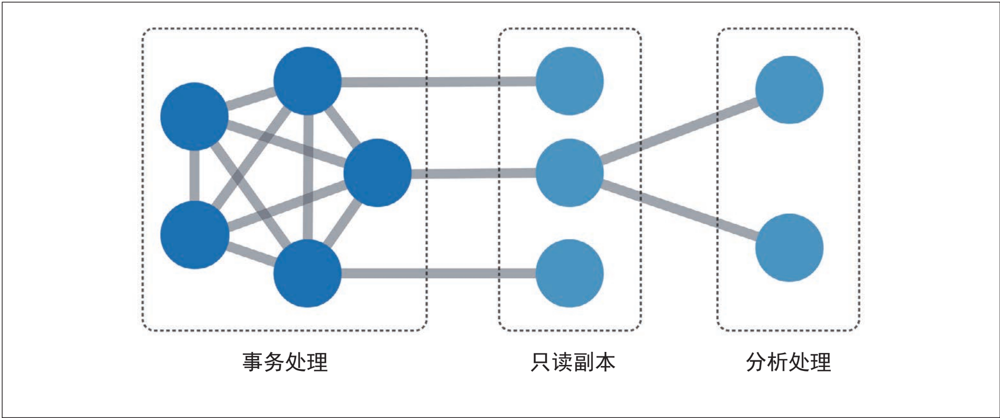


图 1-5：事务处理需要一种支持低延迟查询处理和高数据完整性的混合平台，同时需要在海量数据之上集成复杂分析

Gartner 认为：

> （HTAP）可能会重新定义一些业务流程的执行方式，因为实时高级分析（例如规划、预测和假设分析）变成了处理过程的组成部分，而不再是事后执行的单独活动。这将支持新型的实时业务驱动决策过程。最终，HTAP 将成为面向智能业务操作的关键支持架构。

随着 OLTP 和 OLAP 逐渐集成，开始支持以前仅在单个"竖井"中提供的功能，不再需要为工作载荷使用不同的数据产品或系统——可以使用相同的平台以简化架构。这意味着分析查询可以利用实时数据，而且可以简化分析的迭代过程。

1.4　为何要关心图算法

图算法有助于理解关联数据。关系广泛存在于现实世界的系统中，从蛋白质的相互作用到社交网络，从通信系统到电网，从零售体验到火星任务规划。理解网络及其内部联系为洞察和创新提供了不可思议的潜力。

图算法特别适用于理解结构和揭示高度关联的数据集中的模式。没有什么比大数据更能体现关联性和交互性了。大数据汇集、混合和动态更新的信息量令人瞩目。这正是图算法的用武之地，它有助于我们理解海量数据，针对关系进行更复杂的分析，还可以为人工智能丰富上下文信息。

随着数据之间的联系越来越紧密，理解数据之间的依赖关系变得越来越重要。研究网络增

长的科学家注意到，连接性会随着时间的推移而增强，但是并不均匀。择优连接理论是研究动态增长如何影响结构的一种理论。如图 1-6 所示，该理论描述了这样一种趋势：一个节点倾向于连接到已经有很多连接的节点。

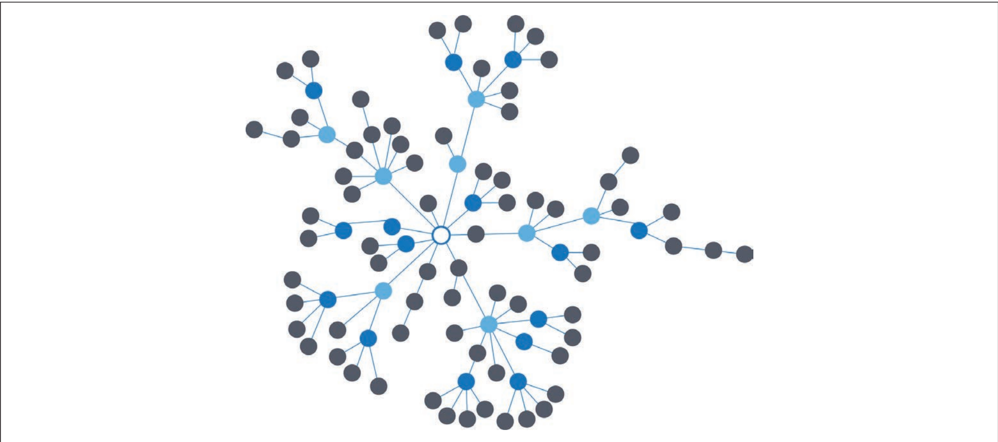


图 1-6：择优连接现象：一个节点的连接关系越多，就越有可能建立新的联系。这导致了不均匀的集中度和中心

Steven Strogatz 的著作《同步：秩序如何从混沌中涌现》给出了一些示例，解释了现实系统进行自组织的不同方式。不管潜在原因如何，许多研究人员相信，网络的增长方式与其最终形态和层级是密不可分的。高密度群组和块状数据网络会不断发展，复杂性会随着数据规模的增长而增长。从互联网到图 1-7 所示的游戏社区社交网络，在当今现实世界的大多数网络中存在这种关系的聚集现象。

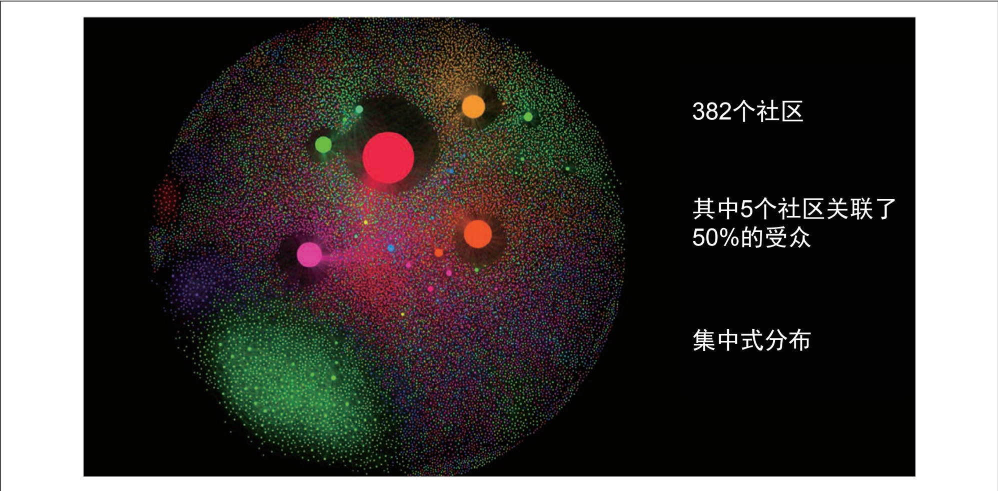


图 1-7：对游戏社区的分析显示，在 382 个社区中，只有 5 个社区的连接最为集中

图 1-7 所示的网络分析是由 Pulsar 公司的 Francesco D’Orazio 提出的，旨在辅助预测视频的病毒式传播趋势并为制定视频分发策略提供信息。Francesco D’Orazio 发现，社区分布的集中度和内容扩散速度存在相关性。

这与平均分布模型预测的结果截然不同。在平均分布模型中，大多数节点具有相同数量的连接。如果万维网的连接是平均分布的，那么所有网页的出入链接数量都相同。平均分布模型认为，大多数节点是平等连接的，但是许多类型的图和许多真实网络表现出集中式分布特征。与旅行网和社交网络的图一样，Web 也具有**幂律分布**特征，即少数节点高度连接，而多数节点的连接较少。

幂　　律

幂律（也称**标度律**）描述了两个量之间的关系，其中一个量随另一个量的幂而变化，例如一个立方体的体积是其边长的 3 次幂。一个著名的例子是**帕累托分布**（也称**二八定律**），它最初用来描述 20% 的人口控制 80% 的财富。在自然界和网络中存在各种幂律。

试图使网络均匀分布的想法通常不适用于研究关系或进行预测，因为现实网络中的节点和关系分布并不均匀。如图 1-8 所示，对不均匀的数据使用平均特征会导致错误的结果。

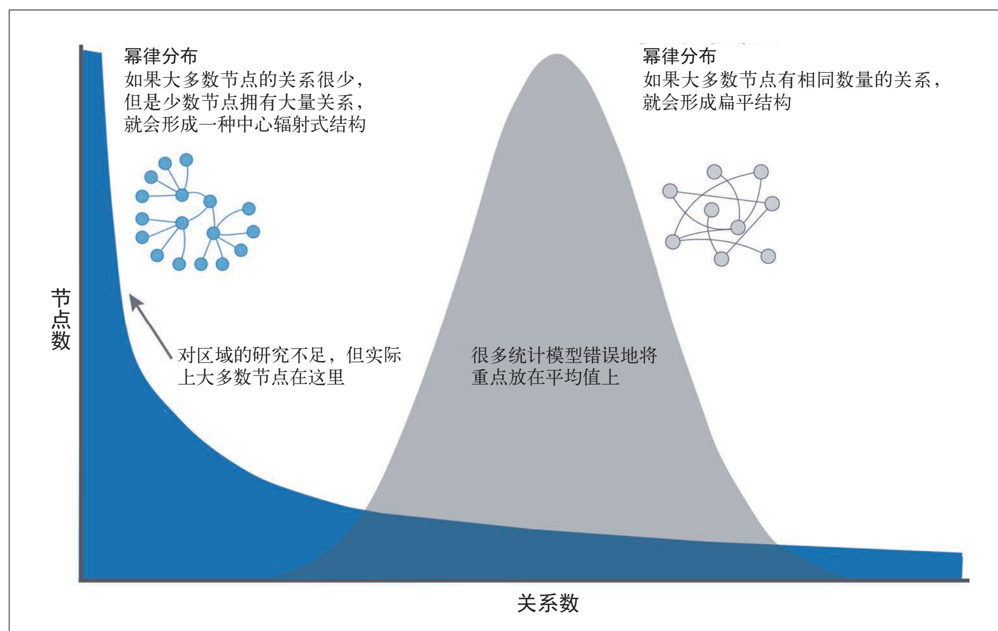


图 1-8：在现实网络中，节点和关系的分布不均匀，在极端情况下表现出幂律分布特征。均匀分布假设大多数节点具有相同数量的关系，这样产生的是一个随机网络

由于高度连接的数据并不遵循平均分布，因此网络科学家使用图分析来搜索和解释真实数据的结构和关系分布。

> 在已知的自然界中，没有任何一个网络可以用随机网络模型来描述。
>
> ——Albert-László Barabási，美国东北大学复杂网络研究中心主任，著有多本网络科学相关图书

大多数用户面临的挑战是，使用传统分析工具来分析密集而不均匀的关联数据非常麻烦。在数据中可能存在某种结构，但是很难找到。采用平均方法处理混乱的数据确实很诱人，但是这种做法会隐藏模式，而且结果无法代表任何真实群组。如果将所有客户的人口统计信息都做平均化处理，并且仅基于平均化处理结果来提供客户体验，那么肯定会漏掉大多数群体：社团往往是围绕年龄、职业、婚姻状况和地址等因素形成的。

此外，通过快照无法发现动态行为，尤其是那些围绕突发性事件和爆发性事件的动态行为。如果社交群体的人际关系不断增加，就意味着有更多的交流。这会导致出现协调临界点，随后出现联盟，或者形成子群体并出现两极分化（例如选举）。预测网络随时间的演变需要更复杂的方法，但是如果了解数据中的结构和交互，就能推断行为。由于图分析注重关系研究，因此也用于预测群组的弹性。

1.5 图分析用例

在最抽象的层次上，图分析可以应用于预测动态群组的行为和规定动作。要实现这一点，就需要了解群组内部的关系和结构。图算法通过检查网络连接的整体特性来实现这一点。通过这种方法，就可以理解关联系统的拓扑结构并为其流程建模。

关于是否需要采用图分析和图算法，图 1-9 列出了 3 类常见问题。

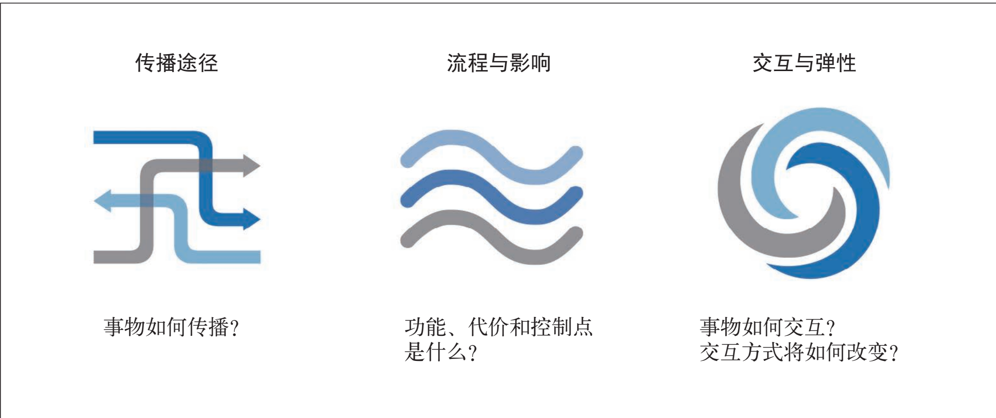


图 1-9：图分析所回答的问题类型

下面是图分析和图算法的一些用例。你所面临的挑战与之相似吗？

- 调查传染病或级联传输故障的传播路径。
- 发现网络中最易受攻击或最易损坏的组件。
- 确定在传送信息或资源时速度最快、代价最小的方式。
- 预测数据中缺失的联系。
- 定位复杂系统中的直接影响和间接影响。
- 发现不可见的层级结构和依赖关系。
- 预测群组将合并还是分裂。
- 发现瓶颈及有权拒绝或提供更多资源的个体。
- 基于行为发现社团以进行个性化推荐。
- 减少欺诈和异常检测中的假正例。
- 为机器学习提取更多预测性特征。

1.6　小结

本章介绍了当今数据是如何紧密关联在一起的，以及其意义所在。在分析群组动态性和关系方面，目前已经有了活跃的科学实践，但是相关工具在很多行业不常见。在评估先进的分析技术时，应该考虑数据的本质特性，思考是否需要了解社团属性或预测复杂行为。如果数据用于表示某个网络，就应该避免这样做，不要缩减要素以采用平均模型，而应该使用与数据和所寻求的见解相匹配的工具。

第 2 章将介绍图的概念和术语。

第 2 章

图论及其概念

本章介绍全书内容设置、图算法相关术语和图论的基础知识，重点解释与图论研究人员关系最密切的那些概念。

首先介绍图的表示方法，然后介绍不同类型的图及其性质。这些内容非常重要，因为图的特征决定了对算法的选择，也有助于理解算法的结果。本章最后将概述本书拟详细介绍的各种图算法类型。

2.1 术语

标记属性图是一种流行的图数据建模方法。

标签将节点标记为群组的一部分。图 2-1 有两组节点：`Person` 和 `Car`。（在经典图论中，标签仅适用于单个节点，而现在通常可用于表示一组节点。）关系可按照**关系类型**进行分类。本例中的关系类型有 `DRIVES`、`OWNS`、`LIVES_WITH` 和 `MARRIED_TO`。

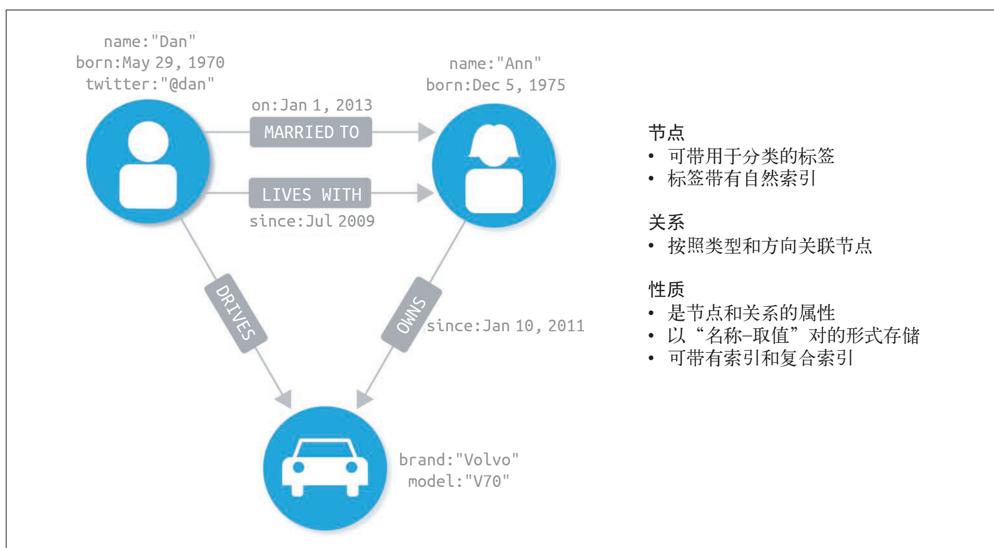


图 2-1：标记属性图模型能够灵活、简洁地表示关联数据

性质（property）是属性（attribute）的同义词，可以包含多种数据，如数值、字符串以及空间数据和时态数据。图 2-1 将性质指定为“名称 – 取值”对，其中性质的名称在前，取值在后。例如左边 `Person` 节点的性质 `name` 取值为 `Dan`，`MARRIED_TO` 关系的性质 `on` 则取值为 `Jan 1, 2013`。

子图是位于较大图内的图[1]。当需要用具有特定特征的子集进行重点分析时，将子图当作过滤器非常有用。

路径由一组节点及其连接关系构成。以图 2-1 为例，一条简单的路径可以包含节点 `Dan`、`Ann` 和 `Car` 以及关系 `DRIVES` 和 `OWNS`。

图的类型、形状、大小以及可用于分析的属性类型都各不相同。下面介绍几种适合图算法处理的图。请记住，这些对图的说明也适用于子图。

2.2　图的类型和结构

在经典图论中，**图**等同于**简单图**（也称**严格图**），其节点之间只有一个关系，如图 2-2 中的左侧例图所示。然而，在大多数实际的图中，节点之间有多个关系，甚至有自引用关系。现在，**图**这个术语通常涵盖了图 2-2 中的 3 种，在本书中亦然。

注 1：更严谨的表述是：子图的节点集和边集分别是原图节点集和边集的子集。——译者注

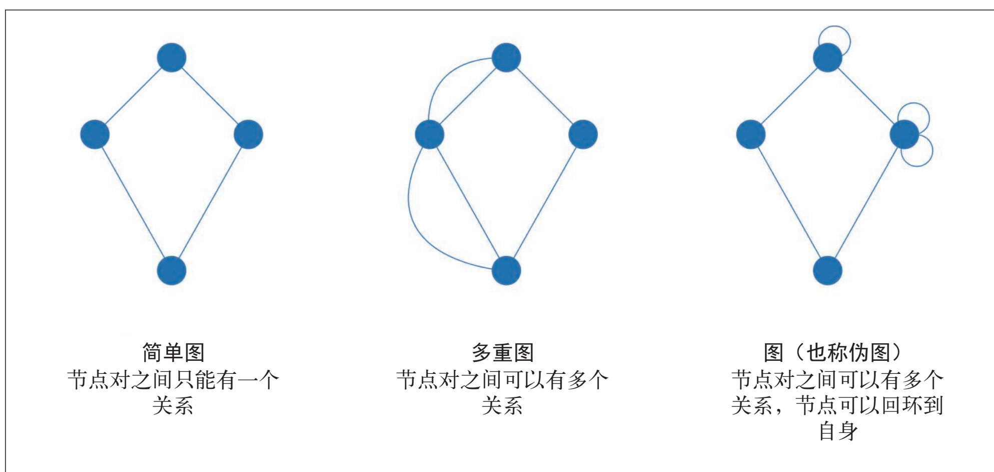


图 2-2：在本书中，图这个术语涵盖 3 种经典的图

随机结构、小世界结构和无标度结构

图可以呈现为多种形状。图 2-3 描绘了 3 种有代表性的网络。

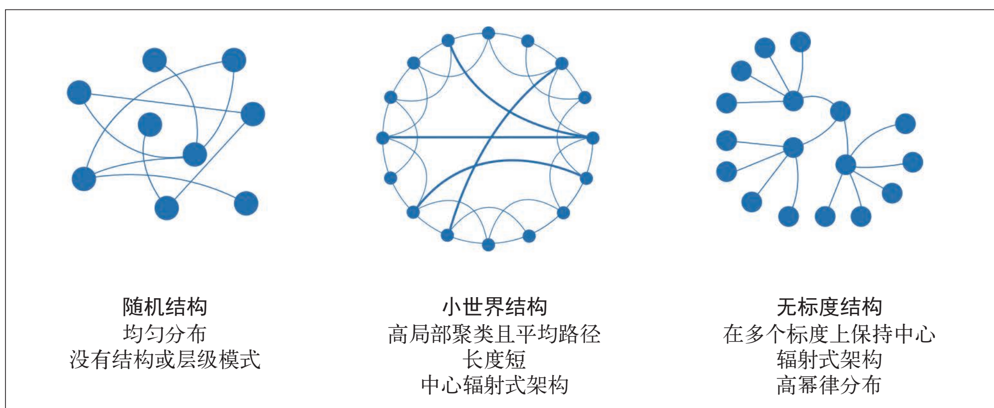


图 2-3：3 种图形不同且行为特征各异的网络结构

- **随机网络**

  随机网络没有层级结构，其连接分布完全平均。这种具有随机形状的图是“扁平的”，且没有可辨识的模式。任一节点连接到其他节点的概率都相同。

- **小世界网络**

  小世界网络在社交网络中极为常见，它展现了局部化的连接和中心辐射模式。Kevin Bacon 的“六度空间”游戏可能是小世界效应最著名的例子。虽然个人经常交往的朋友只是一个小群体，但离任何人都不远——无论是著名演员，还是远在地球另一端的人们。

❑ **无标度网络**

当存在幂律分布时就会产生无标度网络，而且不管标度如何，都会保持中心辐射式架构，例如万维网。

这些网络类型组成了具有独特结构、分布和表现的图。当使用图算法时，结果会包含类似模式。

2.3 图的种类

要想充分利用图算法，熟悉最具代表性的图很重要。表 2-1 总结了图的常见属性，之后详细探讨不同种类的图。

表2-1：图的常见属性

图 属 性	关键因素	算法考虑
连通与不连通	图中任意两个节点之间是否存在一条路径，不考虑距离因素	节点“孤岛”可能会导致意外表现，例如无法处理不连通分量
加权与不加权	关系或节点是否有（特定领域的）值	许多算法涉及权重，如果忽略权重，就会发现算法在性能和结果上都存在显著差异
有向与无向	关系是否明确定义了起始节点和终止节点	这为推断额外含义增加了丰富的上下文信息。在某些算法中，可以明确设定使用单向、双向还是无向
有环与无环	路径的起点和终点是否为同一节点	有环图很常见，但是在算法处理上必须小心（通常按照存储的遍历状态来处理），否则循环可能无法终止。无环图（或生成树）是许多图算法的基础
稀疏与稠密	关系数与节点数的比值	极为稀疏和极为稠密的连通图都会导致异常结果，假如稀疏性或稠密性并非该领域的固有特征，则可以借助数据建模
单部、二部与 k 部	节点只与一种其他类型的节点连接（例如用户喜欢某些电影），或者与其他多种节点连接（用户喜欢爱好某些电影的用户）	有助于创建关系来分析和投影更有用的图

2.3.1 连通图与非连通图

如果图的所有节点之间都存在路径，则该图是连通图。如果图中存在“孤岛”，则为非连通图。如果这些“孤岛”中的节点是连通的，则称其为**分量**（也称**簇**），如图 2-4 所示。

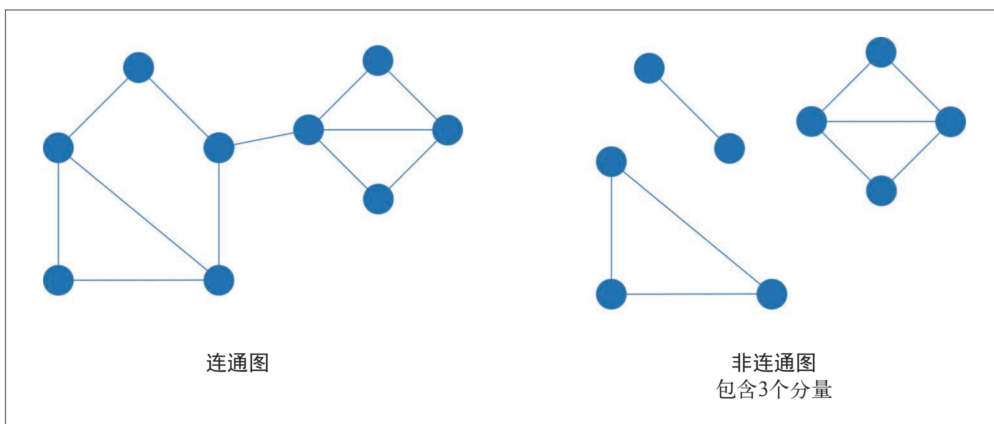


图 2-4：如果图中存在“孤岛”，就是非连通图

有些算法难以处理非连通图，而且可能产生误导性结果。如果出现意外结果，应当首先检查图的结构。

2.3.2　无权图与加权图

无权图没有为其节点或关系赋予权重。对于加权图，这些值可以代表各种量，比如成本、时间、距离、容量，甚至特定领域的优先级。图 2-5 展示了无权图和加权图的差异。

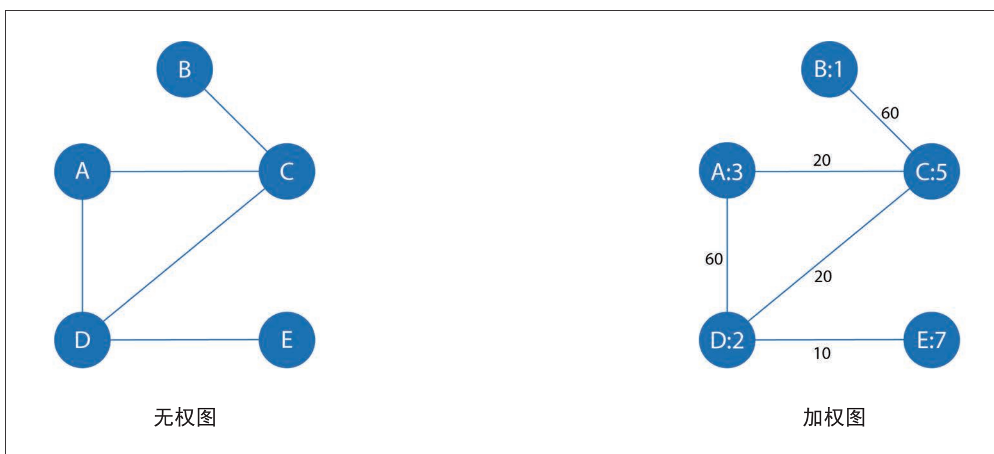


图 2-5：加权图可以保存关系或节点的值

基础的图算法在处理过程中可以使用权重表示关系的强度或取值。许多算法会计算指标，并将其作为进一步处理的权重。有些算法会在进一步寻找累计总数、最小值或最优值时更新权重。

路径查找算法是加权图的经典应用之一。这种算法支撑着手机上的地图应用程序，它们能计算出各位置之间最短、最经济或最快的交通路线，例如图 2-6 使用两种方法来计算最短路径。

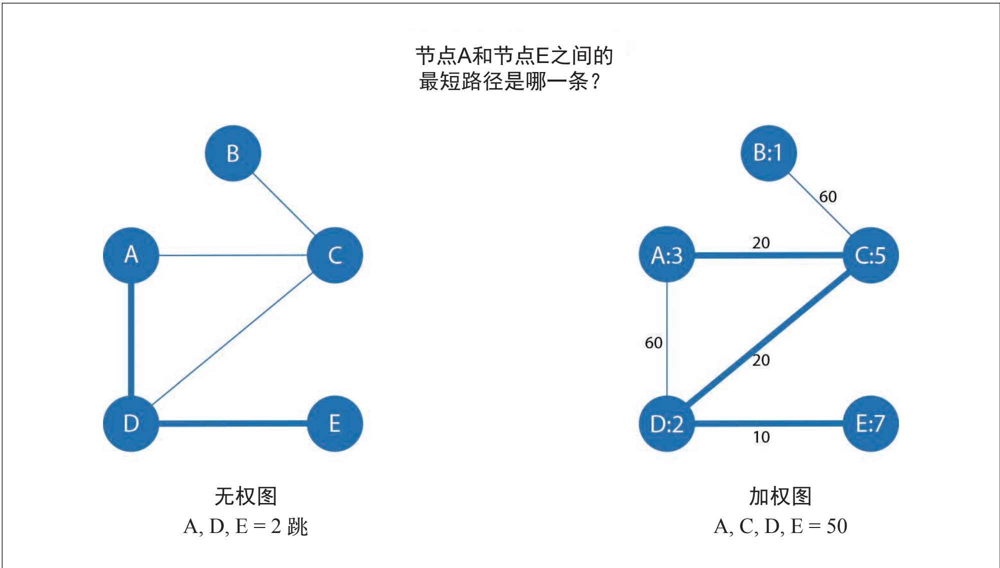


图 2-6：看似相同的无权图和加权图，最短路径可能不同

当没有权重时，最短路径是根据关系（通常称为**跳**）的数量计算的。A 和 E 之间有一条两跳的最短路径，这说明它们之间只有一个节点（D）。从 A 到 E 的最短加权路径则是从 A 到 C 到 D 再到 E。在这种情况下，以跳数计算的最短路径实际上是物理距离为 70 千米的那条较长的路径。

2.3.3　无向图与有向图

在无向图中，关系是双向的（例如友谊）；在有向图中，关系有特定的方向。指向某个节点的关系称为**入连接**（in-link）；同理，源于某个节点的关系称为**出连接**（out-link）。

方向增加了另一个维度的信息。类型相同但方向相反的关系具有不同的语义，这是因为方向表达了依赖关系或指明了流程。这种特性也用作可信度或群体力量的指标。借助方向还可以很好地表达个人偏好和社会关系。

假设图 2-7 中的有向图是一个人际网络且关系为“喜欢”，那么通过计算会发现 A 和 C 更受欢迎。

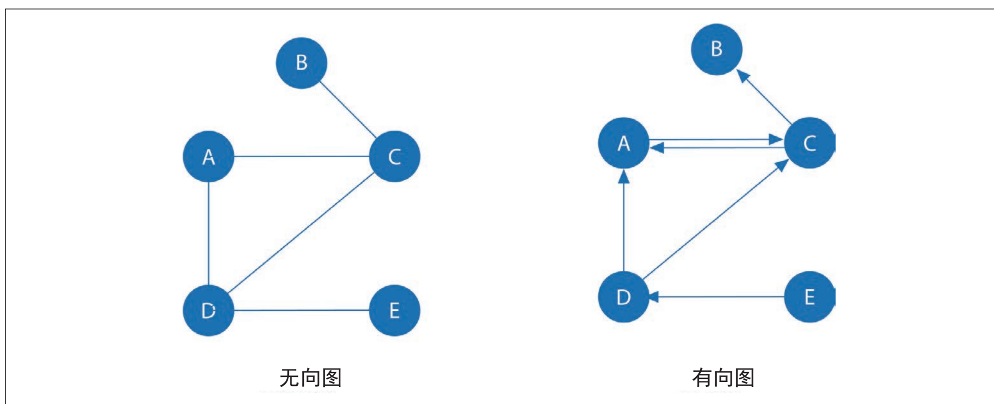


图 2-7：许多算法允许分别基于入连接或出连接、基于双向或不考虑方向进行计算

道路网可以说明这两种图都有用，例如城市之间的高速公路通常是双向的，然而在城市内部，有些道路是单行道（某些信息流也是如此）。

与有向方式相比，以无向方式运行算法会得到不同的结果。在无向图中，例如高速公路或友谊关系等，我们假设所有关系都是双向的。

把图 2-7 重新想象成一个有向公路网，可以从 C 和 D 开车到 A，但是只能经 C 离开。这样一来，如果从 A 到 C 不存在关系，就行不通了。单行道路网一般不太可能出现这种情况，但是对于处理流程或网页来说就不是这样的了。

2.3.4　无环图与有环图

在图论中，从同一节点开始和结束，其间经过的关系和节点所形成的路径就是环，无环图则没有这样的环。如图 2-8 所示，有向图和无向图都可以有环，但对于有向图，路径要沿着关系方向前进，其中图 1 是**有向无环图**（directed acyclic graph，DAG），根据定义，该图总有**端点**（也称**叶节点**）。

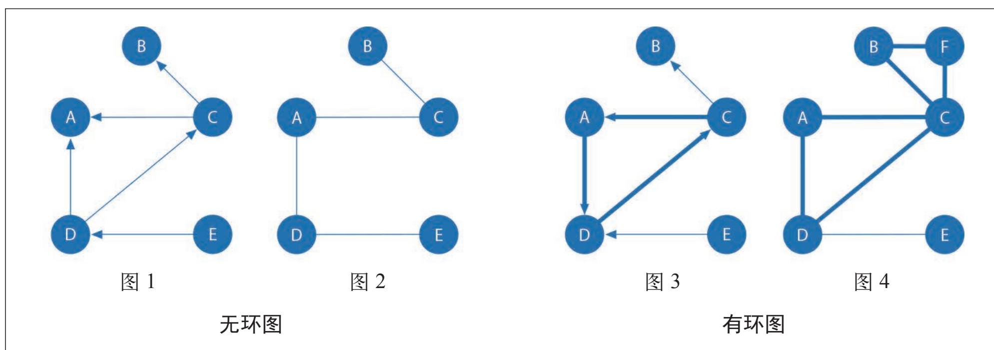


图 2-8：在无环图中，如果不回溯步骤，就不可能从同一节点开始和结束

在图 2-8 中，图 1 和图 2 没有环，这是因为无法在同一节点开始和结束且关系不重复。第 1 章讲过，图论的起源是“哥尼斯堡七桥”问题，该问题所要求的正是关系不重复！图 2-8 中的图 3 展示了一个按照 A-D-C-A 走向的简单环，其中没有重复节点。在图 4 中，添加节点和关系之后，无向有环图变得更有意思了，它形成了带有重复节点（C）的闭环，按照 B-F-C-D-A-C-B 的顺序走向，实际上图 4 中有多个环。

环很常见，有时需要将有环图转换为无环图（通过切割关系）来消除处理方面的问题。在调度、谱系学和版本历史等问题中经常出现有向无环图。

树

在经典图论中，无向无环图也称为**树**。在计算机科学中，树也可以有向。一个更宽泛的定义是：树是任何两个节点间都仅有一条路径连接的图。树对于理解图的结构和许多算法很重要。它们在为改进分类或组织层级而设计网络、数据结构以及最优搜索方案等方面起着关键作用。

有关树及其变体的著作已经有很多了。图 2-9 展示了几种常见的树。

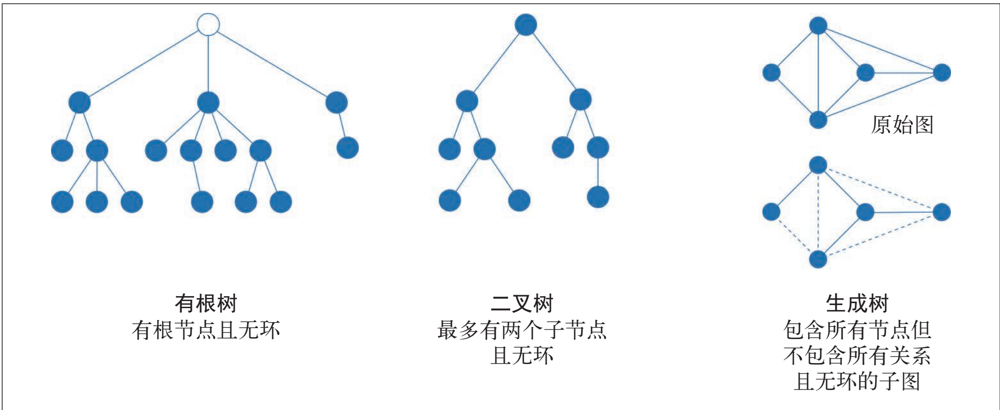


图 2-9：在这些原型树的图中，生成树是更常用的图算法

在这些变体中，生成树与本书内容最为相关。生成树是子图，它包含了一个更大无环图的所有节点，但不包含该图的所有关系。最小生成树用最少的跳数或者说权重最小的路径连接图中的所有节点。

2.3.5　稀疏图与稠密图

图的稀疏性基于图所拥有的关系数与其最大可能关系数的比值来度量，当每对节点之间都存在关系时即可得到最大可能关系数。如果图的每个节点都与其他节点之间存在关系，则称其为**完全图**，或者对分量来说也可称为**团**。如果我所有的朋友都认识对方，就形成了一个“团”。

图的**最大密度**是指其完全图中可能存在的关系数。可以通过公式 $\text{Max}D=\frac{N(N-1)}{2}$ 来计算，其中 N 为节点数。使用公式 $D=\frac{2(R)}{N(N-1)}$ 来度量**实际密度**，其中 R 是关系数。图 2-10 展示了 3 个度量无向图实际密度的例子。

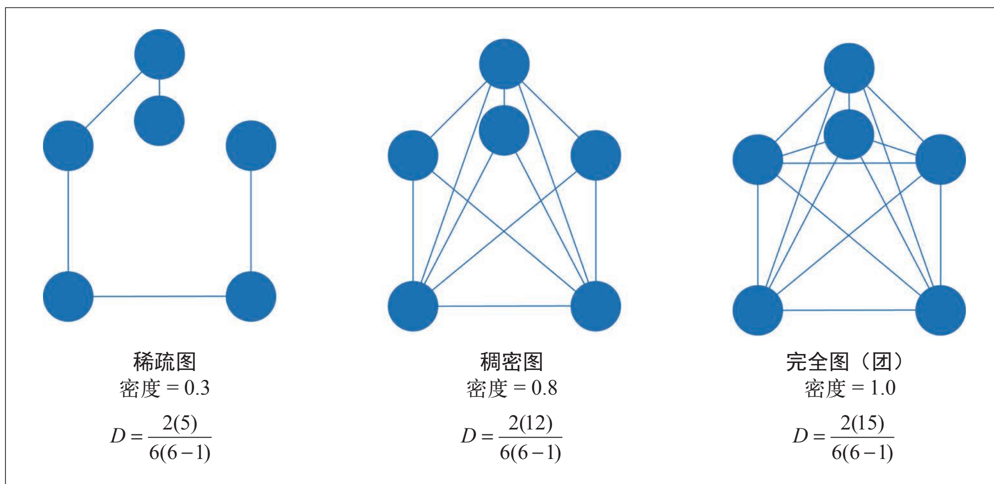


图 2-10：检查图的密度有助于评估意外的结果

虽然没有严格的分界线，但是任何实际密度接近最大密度的图都可以看作稠密图。大多数基于真实网络的图较为稀疏，且节点总数和关系总数之间近似于线性相关，在物理因素发挥作用的情况下更是如此，例如一个节点能够连接多少电线、管道、道路或友谊等都是有实际限制的。

针对极其稀疏或稠密的图执行某些算法会返回无意义的结果。一方面，如果图太稀疏，那么可能没有足够的关系来支撑算法计算出有用的结果。另一方面，在连接非常稠密的节点之间不会有太多额外信息，这是因为它们的关系已经足够紧密了。高密度会扭曲一些结果或者增加计算复杂度。在这些情况下，过滤相关子图是一种实用方法。

2.3.6　单部图、二部图和*k*部图

大多数网络包含具有多种节点和关系的数据，然而图算法通常只考虑一种节点和一种关系。只有一种节点和一种关系的图有时称为**单部图**。

二部图的节点可以划分成两个集合，其关系仅连接一个集合的节点和另一个集合的节点，如图 2-11 所示。该图有两个节点集合：一个观众集合和一个电视剧集合，而且只存在两个集合之间的关系，集合内部不存在连接关系。换言之，在图 2-11 的图 1 中，电视剧彼此不关联，而只与观众相关联。观众彼此也不关联。

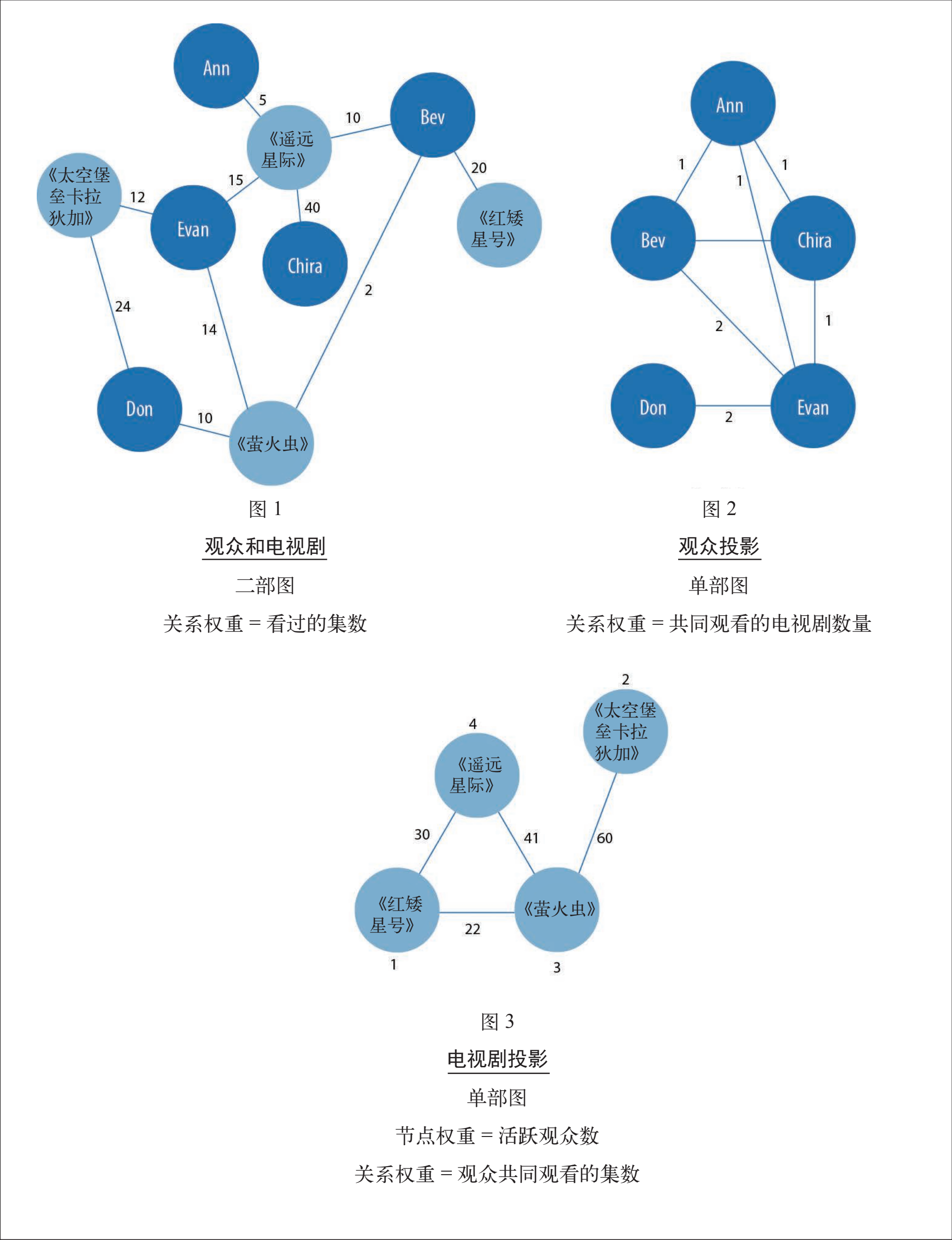


图 2-11：通常把二部图投影为单部图以进行更具体的分析

在由观众和电视剧构成的二部图的基础上，创建了两个单部图投影：基于共同观看的电视剧创建的观众连接图（图 2）和基于共同观众的电视剧连接图（图 3）。还可以基于关系类

型进行过滤，比如观看、评分或评论。

投影出含有推断连接的单部图是图分析的重要组成部分。这些投影类型有助于揭示间接关系及其强度。例如在图 2-11 的图 2 中，Bev 和 Ann 只看过一部共同的电视剧，而 Bev 和 Evan 都看过的有两部。在图 3 中，我们通过共同观众的聚合视图对电视剧之间的关系进行了加权。该指标或相似度等其他指标，可用于推断观看《太空堡垒卡拉狄加》和《萤火虫》的活动之间存在什么内在联系。之后就可以向类似于 Evan 这样（刚刚看完《萤火虫》最后一集）的观众发送推荐消息。

***k* 部图**涉及的节点类型数为 *k*，例如三部图有 3 种节点。这只是对二部图和单部图的扩展，使之用于多节点类型。现实世界中的许多图，尤其是知识图谱，其 *k* 值往往很大，这是因为它们整合了许多概念和信息类型。以设计新配方为例，它是将配方集映射到配料集再映射到化合物，然后推导出关联大众偏好的新组合。还可以通过泛化来减少节点类型的数量，将节点的诸多形式泛化为一个节点，例如可以把“菠菜”和“油菜”视为“绿叶蔬菜”。

前面介绍了几种常用的图，下面介绍这些图可以应用的图算法类型。

2.4 图算法的类型

下面探究在图算法中处于核心地位的 3 个分析领域。这 3 个类别分别对应第 4 章、第 5 章和第 6 章。

2.4.1 路径查找

路径是图分析和图算法的基础，自此开始介绍具体的算法示例。查找最短路径是使用图算法执行的相当频繁的任务，而且它还是几种分析的先驱。最短路径是跳数最少或权重最小的遍历路径。如果图是有向的，它就是指两个节点之间关系方向所允许的最短路径。

> **路径类型**
>
> **平均最短路径**用于考察网络的整体效率和弹性，比如了解各个地铁站之间的平均距离，有时可能还想了解站点间的最远优化路线，比如确定哪些地铁站之间的距离最远，或者站点之间（即使选择最佳路线）的站数最多。对这种情况，我们将使用图的**直径**来查找所有节点对之间的最长路径或最短路径。

2.4.2 中心性

中心性的要点就是了解网络中哪个节点更重要，但是这里所说的“重要”是什么含义呢？人们创建了不同类型的中心性算法来度量不同的事物，例如快速传播信息的能力和桥接不同群体的能力。本书将重点讨论节点和关系的组织方式。

2.4.3 社团发现

连通性是图论的核心概念之一，它支持复杂网络分析，比如社团发现。现实世界中的大多数网络或多或少呈现出独立子图这样的子结构（通常是准分形）。

连通度用于发现社团并且量化分组的质量。评估图中不同类型的社团有助于揭示图的结构，比如中心结构和层级结构，也有助于了解某个群组吸引或排斥其他群组的倾向。这些方法用于研究一些突发现象，例如那些导致回音壁效应和过滤气泡效应的现象。

2.5 小结

图是很直观的，而且它与我们思考和绘制系统的方式一致。当理解了相关术语和知识层次后，很快就能掌握图的精髓。本章介绍了后续要用到的概念、表示方式，以及将谈到的各种图。

图论参考书目

若有兴趣学习更多图论相关知识，推荐阅读如下入门教材。

- *Introduction to Graph Theory*，Richard J. Trudeau 著，这是一本入门书，写得很好且容易理解。
- 《图论导论（第 5 版）》，Robin J. Wilson 著，这是一本入门书，图文并茂。
- *Graph Theory and Its Applications, Third Edition*，Jonathan L. Gross、Jay Yellen 和 Mark Anderson 合著，该书提供了很多习题，讲解细致。读者需要有一定的数学功底。

接下来，在深入研究如何在 Apache Spark 和 Neo4j 平台上应用图算法之前，首先了解图处理和分析类型。

第3章

图平台和图处理

本章将简要介绍图处理的各种方法以及常用平台，进一步研究本书所用的两个平台：Apache Spark 和 Neo4j，以及它们的适用场景。此外，本章还将提供两个平台的安装指南，以便为后续几章做好铺垫。

3.1 图平台和图处理的注意事项

图处理具有一些独特的性质，例如其计算过程是由结构驱动、全局聚焦的，而且很难解析。本节将介绍图平台和图处理的一般注意事项。

3.1.1 平台注意事项

对于图处理，纵向扩展和横向扩展哪个更好，一直存在争议。是应该使用强大的多核、大内存计算机，并且关注高效的数据结构和多线程算法，还是应该在分布式处理框架和与之相关的算法上投入精力呢？

COST（configuration that outperforms a single thread）是一种很有用的评估方法，Frank McSherry、Michael Isard 和 Derek Murray 的论文“Scalability! But at What COST?”有相关介绍。COST 可用于比较系统的可伸缩性和引入的开销。其核心思想是，配置良好的系统使用优化的算法和数据结构，在性能上能超过当前通用的横向扩展解决方案。采用这种方法衡量性能时，并不会出现由于采用并行处理而掩盖低效率的情况。将可伸缩性和高效使用资源的概念区分开来，有助于构建可直接按需配置的平台。

有些图平台包含了高度集成的解决方案，优化了算法、处理和内存检索，使之能够紧密协调工作。

3.1.2 处理注意事项

处理数据的方式有多种，例如针对基于记录的数据，可以采用流处理或批处理，抑或采用 MapReduce 方式；而对图数据而言，目前也有一些方法可将图结构中的固有数据依赖整合到处理中。

- **节点中心式**

  这种方式将节点作为处理单元，保存节点的累加结果和计算状态，并通过消息将通信状态变化情况传递给其邻节点。这种模型使用现成的转换函数可以更简单地实现每种算法。

- **关系中心式**

  这种方式与节点中心式模型有相似之处，但它也可用于子图和序贯分析。

- **图中心式**

  采用这种方式的模型在处理某个子图中的节点时独立于其他子图，通过消息传递与其他子图进行通信（极少）。

- **遍历中心式**

  采用这种方式的模型把遍历器在图上穿行时积累的数据用于计算。

- **算法中心式**

  这种方式使用各种方法优化每个算法的实现。这是前述几种模型的混合模式。



Pregel 是由谷歌公司创建的一种容错式并行处理框架，以节点为中心，可用于分析大型图的性能。Pregel 基于整体同步并行模型（bulk synchronous parallel model，以下简称 BSP 模型），而 BSP 模型具有独特的计算阶段和通信阶段，以此简化并行编程。

Pregel 在 BSP 模型之上添加了一个节点中心式抽象层，算法可以借此计算由每个节点的邻节点通过消息传入的值。每次迭代都会执行一次这样的计算，之后可以更新节点值并向其他节点发送消息。在通信阶段，节点还可以组合要传输的消息，这有助于减少网络通信量。当不再发送新的消息或达到某一设定边界后，算法执行结束。

大多数专用于图的方法需要加载整张图以实现高效的跨拓扑操作，这是因为分割和分发图数据会导致大量数据传输，造成工作实例重新配置，而这对于很多需要迭代处理全局图结构的算法而言是比较困难的。

3.2 典型平台

为满足图处理需求，已经出现了几种平台。图计算引擎和图数据库通常是分离的，这就要求用户根据其处理需求来迁移数据。

- **图计算引擎**

  图计算引擎是只读的非事务处理引擎，主要用于高效执行图的迭代分析和对整张图的查询。图计算引擎支持图算法的各种定义和处理样式，比如节点中心式方法（如 Pregel、GAS[1] 模型）或基于 MapReduce 的方法（如 PACT），Giraph、GraphLab、Graph-Engine 和 Apache Spark 属于此类引擎。

- **图数据库**

  从事务处理的背景来看，图数据库主要关注用小型查询进行快速读写，通常只涉及图的一小部分。图数据库的优点是操作稳健性和面向多用户的高并发可伸缩性。

3.2.1 选择平台

选择生产平台需要考虑很多因素，比如要运行的分析类型、性能需求、现有环境以及团队偏好等。本书使用 Apache Spark 和 Neo4j 来展示图算法，因为它们都具有一些独特优势。

Apache Spark（以下简称 Spark）是一个横向扩展和以节点为中心的图计算引擎。其流行计算框架和库支持数据科学的多种工作流程。Spark 平台适用于如下场景。

- 算法本质上是可并行或可分割的。
- 算法工作流程需要在多种工具和多种语言的环境中进行“多语言”操作。
- 分析可以以批处理模式离线运行。
- 图分析针对没有转换成图格式的数据。
- 团队需要自己编写和实现算法，并且具备相关的专业知识。
- 团队使用图算法的频率不高。
- 团队更喜欢在 Hadoop 生态系统中保存所有数据和分析结果。

Neo4j 图平台是图数据库和算法中心式处理紧密集成的典范，针对图进行了优化。该平台在构建基于图的应用方面广受欢迎，而且提供了与自带图数据库配套调优的图算法库。Neo4j 适用于下述场景。

- 算法需要更多迭代，并且需要良好的内存局域性。
- 算法和结果对性能敏感。
- 图分析针对的是复杂的图数据，或需要进行深度路径遍历。

注 1：GAS 模型包含 gather-apply-scatter 这 3 个阶段。——译者注

- 分析结果与事务性工作负载集成。
- 结果用于增强已有的图。
- 团队需要集成基于图的可视化工具。
- 团队希望采用预打包和提供支持的算法。

此外，有些组织兼用 Spark 和 Neo4j 进行图处理：将 Spark 用于大规模数据集的高层筛选与预处理以及数据集成，将 Neo4j 用于特定处理并且与基于图的应用集成。

3.2.2 Apache Spark

Spark 是用于大规模数据处理的图计算引擎。它使用一种名为 DataFrame 的抽象表来表示和处理以行列（列要指定名称和类型）方式组织的数据。该平台集成了多种数据源，支持 Scala、Python 和 R 等语言。Spark 支持多种分析库，如图 3-1 所示。它具备基于内存的系统，在运行时采用高效分布式计算图。

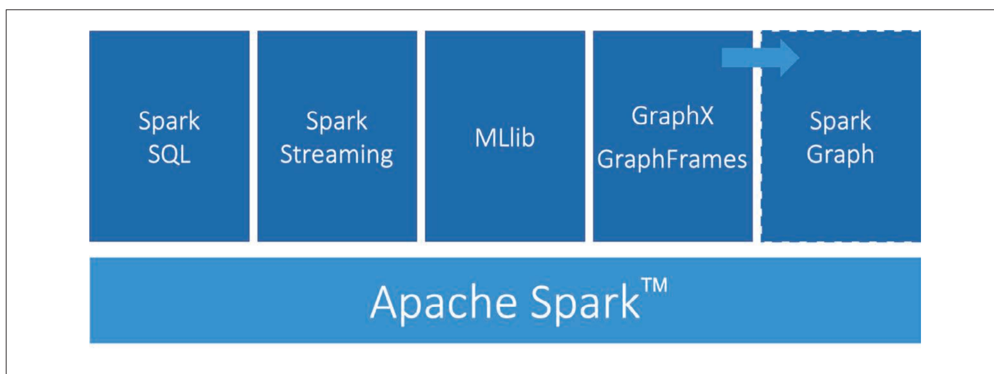


图 3-1：Spark 是一个开源的分布式通用集群计算框架，包含适用于各种工作负载的模块

GraphFrames 是 Spark 的一个图处理库，于 2016 年替代 GraphX，但是它与核心 Spark 是分离的。GraphFrames 基于 GraphX，但是使用 DataFrame 作为底层数据结构。GraphFrames 支持 Java、Scala 和 Python 等编程语言。2019 年春，提案“Spark Graph: Property Graphs, Cypher Queries, and Algorithms”被接受（参见本章“Spark Graph 的发展”部分的说明）。我们希望使用 DataFrame 框架和 Cypher 查询语言，将许多图功能引入核心 Spark 项目中。此外，本书的示例基于 Python API（PySpark），这是因为它广受 Spark 数据科学家欢迎。

Spark Graph 的发展

Spark Graph 项目是由来自 Databricks 和 Neo4j 的 Apache 项目贡献者共同发起的，目的是在核心 Spark 项目的 3.0 版本中增加对 DataFrame、Cypher 和基于 DataFrame 的算法的支持。

Cypher 最初是在 Neo4j 中实现的一种声明式图查询语言，但是通过 openCypher 项目，它现已被多家数据库厂商以及开源项目 Cypher for Apache Spark（CAPS）所使用。

希望在不久的将来能够使用 CAPS 加载和查询图数据，将其作为 Spark 平台的集成部件之一。我们将在 Spark Graph 项目完成后发布 Cypher 示例。

这些进展并不会影响本书所介绍的算法，但可能会添加新的程序调用方式。图算法的底层数据模型、概念和计算方式将保持不变。

节点和关系都可以表示为 DataFrame 对象，且每个节点都有唯一的 ID，而每个关系都有源节点和目标节点。表 3-1 给出了节点 DataFrame 示例，表 3-2 给出了关系 DataFrame 示例。基于这两个 DataFrame 对象的 GraphFrame 包含 JFK 和 SEA 两个节点，以及一个从 JFK 到 SEA 的关系。

表3-1：节点DataFrame

id	city	state
JFK	New York	NY
SEA	Seattle	WA

表3-2：关系DataFrame

src	dst	delay	tripId
JFK	SEA	45	1058923

节点 DataFrame 必须具有 `id` 列——该列的取值用于唯一标识每个节点。关系 DataFrame 必须具有 `src` 列和 `dst` 列——这两列的取值描述该关系连接了哪些节点，并且应该参照节点 DataFrame 的 `id` 列中的记录。

可以使用任何 DataFrame 数据源（包括 Parquet、JSON 和 CSV）来加载节点 DataFrame 和关系 DataFrame，而查询可以组合使用 PySpark API 和 Spark SQL 来进行描述。

GraphFrames 库还为用户提供了实现非开箱即用算法的扩展点。

安装 Spark

可以从 Apache Spark 网站下载 Spark。下载完成后，需要安装以下库才能通过 Python 执行 Spark 作业。

```
pip install pyspark graphframes
```

然后执行以下命令启动 pyspark REPL[2]：

注 2：REPL 即 Read-Eval-Print Loop，是一个简单的交互式编程环境。——译者注

```
export SPARK_VERSION="spark-2.4.0-bin-hadoop2.7"
./${SPARK_VERSION}/bin/pyspark \
  --driver-memory 2g \
  --executor-memory 6g \
  --packages graphframes:graphframes:0.7.0-spark2.4-s_2.11
```

撰写本书时，Spark 的最新发布版本是 Spark-2.4.0-bin-hadoop2.7，读者阅读本书时情况可能会有变化。若是如此，请确保正确更改 SPARK_VERSION 环境变量。



尽管应该在计算机集群上执行 Spark 作业，但为了便于演示，我们仅在单机上执行这些作业。Bill Chambers 和 Matei Zaharia 所著的《Spark 权威指南》一书涵盖了在生产环境中运行 Spark 的更多内容。

现在，你已经做好在 Spark 上运行图算法的准备了。

3.2.3 Neo4j图平台

Neo4j 图平台支持图数据的事务处理和分析处理。它提供了图存储与计算功能以及数据管理与分析工具。该集成工具集位于常见协议、API 和查询语言（Cypher）之上，针对不同用途提供高效访问，如图 3-2 所示。

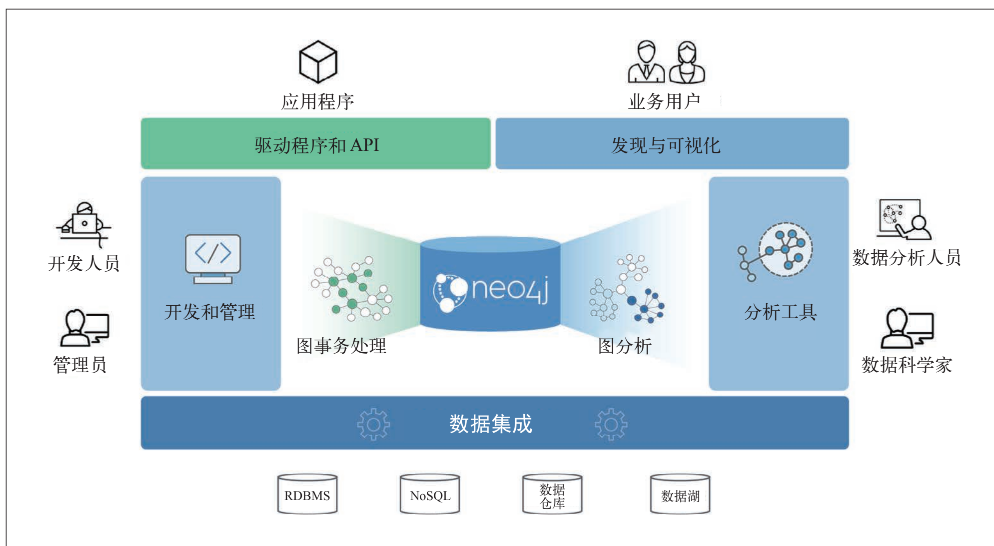


图 3-2：Neo4j 图平台围绕支持事务处理应用与图分析的本地图数据库构建

本书将使用 Neo4j 图算法库。该库以插件形式与数据库一起安装，提供了一个可通过 Cypher 查询语言执行的用户自定义程序集。

该图算法库提供了支持图分析和机器学习工作流程的并行算法。这些算法在基于任务的并

行计算框架上执行，并针对 Neo4j 平台进行了优化。对于不同规模的图，有些组织的内部实现可以纵向扩展到数百亿个节点和关系。

可以将结果以元组流和表格的形式（可用作进一步处理的驱动表）传输到客户端。还可以选择将结果作为节点属性或关系类型高效地回写到数据库。



本书还会用到 Neo4j 的 APOC 库（Awesome Procedures on Cypher）。APOC 库由数百个程序和函数组成，用于辅助完成数据集成、数据转换和模型重构等常规任务。

安装 Neo4j

开发人员操作本地 Neo4j 数据库最便捷的方式是采用 Neo4j Desktop，可以从 Neo4j 官网下载。安装并启动 Neo4j Desktop 之后，便能以插件形式安装图算法库和 APOC 库了。在左侧菜单中创建一个项目并选定，然后在你想要安装插件的数据库上单击 Manage 按钮。在 Plugins 选项卡中有若干插件选项。单击图算法和 APOC 相应的 Install 按钮即可进行安装，参见图 3-3 和图 3-4。

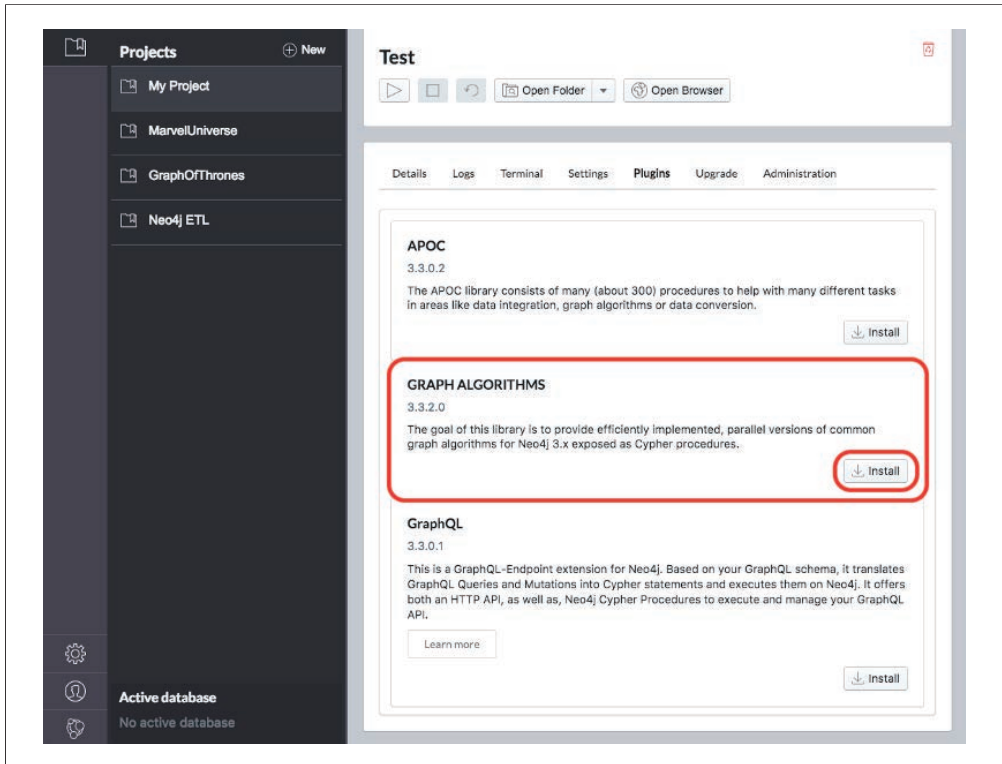


图 3-3：安装图算法库

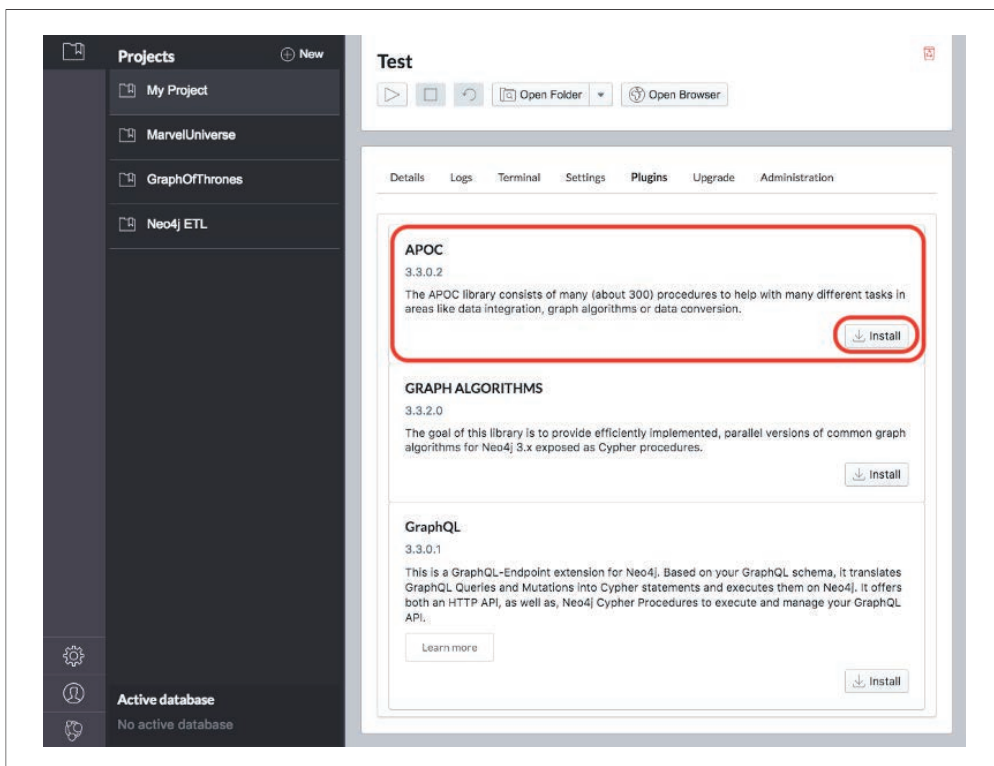


图 3-4：安装 APOC 库

Jennifer Reif 在其博客文章“Explore New Worlds—Adding Plugins to Neo4j”中详细介绍了上述安装过程。现已准备就绪，稍后会介绍如何在 Neo4j 中运行图算法。

3.3 小结

前面介绍了图分析对研究现实网络的重要性，以及图的基本概念、分析和处理，这些内容为理解如何运用图算法奠定了坚实基础。第 4 章将通过 Spark 示例和 Neo4j 示例来讲解如何运行图算法。

第 4 章

路径查找算法和图搜索算法

图搜索算法对图的探究既可以是一般性发现，也可以是显式搜索。这些算法在图中构造路径，但并不要求这些路径的计算性能最强。本章将介绍广度优先搜索和深度优先搜索，它们既是遍历图的基础，也往往是各种分析的首要步骤。

路径查找算法基于图搜索算法，探索节点之间的路径，从某个节点开始遍历关系，直到抵达目的地。这些算法用于识别图中的最优路径，可用于物流规划、最低成本呼叫或 IP 路由、模拟博弈等。

具体来说，本章将介绍如下路径查找算法。

- **最短路径算法以及两个有用的变体（A* 算法和 Yen 算法）**
  用于查找两个选定节点之间的最短路径。
- **所有点对最短路径算法和单源最短路径算法**
  用于查找所有节点对之间的最短路径或从选定节点到其他各节点的最短路径。
- **最小生成树算法**
  用于查找连通的树结构，保证从选定节点访问其他各节点的代价最小。
- **随机游走算法**
  这是对机器学习工作流程和其他图算法都很有用的预处理或采样步骤。

本章将介绍这些算法的运作原理，并给出相应的 Spark 示例和 Neo4j 示例。当算法仅在一种平台上可用时，则仅提供一个示例或说明如何自定义实现。

图 4-1 展示了这几种算法之间的关键区别，表 4-1 是每种算法及其示范用例的速查表。

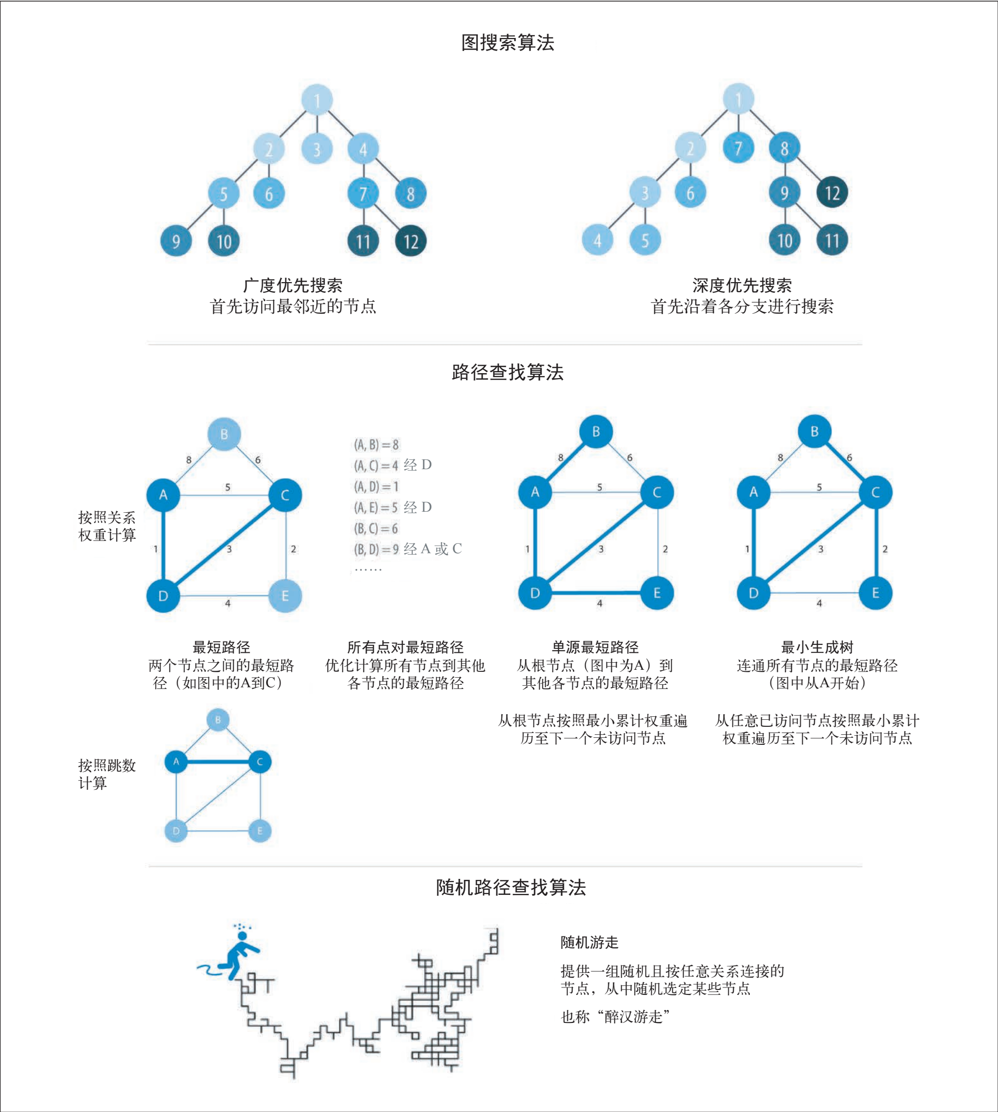


图 4-1：路径查找算法和图搜索算法

表4-1：路径查找算法和图搜索算法总览

算法类型	作　　用	示范用例	Spark 示例	Neo4j 示例
广度优先搜索算法	按下述原则遍历树结构：分别探索最邻近的节点，然后探索这些节点的次级邻节点	在 GPS 系统中定位邻节点，以识别附近感兴趣的地点	有	无

（续）

算法类型	作　用	示范用例	Spark 示例	Neo4j 示例
深度优先搜索算法	按下述原则遍历树结构：在回溯之前尽可能远地探索每个分支	在带有分层选择的博弈模拟中发现最优路径	无	无
最短路径算法变体：A* 算法和 Yen 算法	计算一对节点间的最短路径	查找两地间的行车方向	有	有
所有点对最短路径算法	计算图中所有节点对之间的最短路径	在交通拥堵时评估替代路线	有	有
单源最短路径算法	计算单个根节点到其他各节点的最短路径	电话寻呼的最低成本路由	有	有
最小生成树算法	计算构成一棵连通树结构的路径，使访问所有节点的代价最小	优化连通的路线，例如布线或垃圾回收	无	有
随机游走算法	随机选择关系遍历一条指定规模的路径，返回该路径所经过的节点集合	用于为机器学习扩充训练数据或为图算法补充数据	无	有

首先看看示例中要用到的数据集，了解如何将数据导入 Spark 和 Neo4j。对于每种算法，书中都先简要介绍，然后讲解算法的运算过程，还就何时使用相应算法给出指导，最后给出使用样例数据集的示例代码。

下面开始吧！

4.1　示例数据：交通图

所有关联数据都包含节点之间的路径，这正是将搜索和路径查找作为图分析起点的原因。交通数据集直观地展现了这种关系。本章示例采用的图包含欧洲公路网的一个子集。相应的节点文件和关系文件（如表 4-2 和表 4-3 所示）见本书配套文件。

表4-2：transport-nodes.csv

id	latitude	longitude	population
Amsterdam	52.379189	4.899431	821752
Utrecht	52.092876	5.104480	334176
Den Haag	52.078663	4.288788	514861
Immingham	53.61239	-0.22219	9642
Doncaster	53.52285	-1.13116	302400
Hoek van Holland	51.9775	4.13333	9382
Felixstowe	51.96375	1.3511	23689
Ipswich	52.05917	1.15545	133384
Colchester	51.88921	0.90421	104390
London	51.509865	-0.118092	8787892
Rotterdam	51.9225	4.47917	623652
Gouda	52.01667	4.70833	70939

表4-3：transport-relationships.csv

src	dst	relationship	cost
Amsterdam	Utrecht	EROAD	46
Amsterdam	Den Haag	EROAD	59
Den Haag	Rotterdam	EROAD	26
Amsterdam	Immingham	EROAD	369
Immingham	Doncaster	EROAD	74
Doncaster	London	EROAD	277
Hoek van Holland	Den Haag	EROAD	27
Felixstowe	Hoek van Holland	EROAD	207
Ipswich	Felixstowe	EROAD	22
Colchester	Ipswich	EROAD	32
London	Colchester	EROAD	106
Gouda	Rotterdam	EROAD	25
Gouda	Utrecht	EROAD	35
Den Haag	Gouda	EROAD	32
Hoek van Holland	Rotterdam	EROAD	33

图 4-2 展示了我们要构造的目标图。

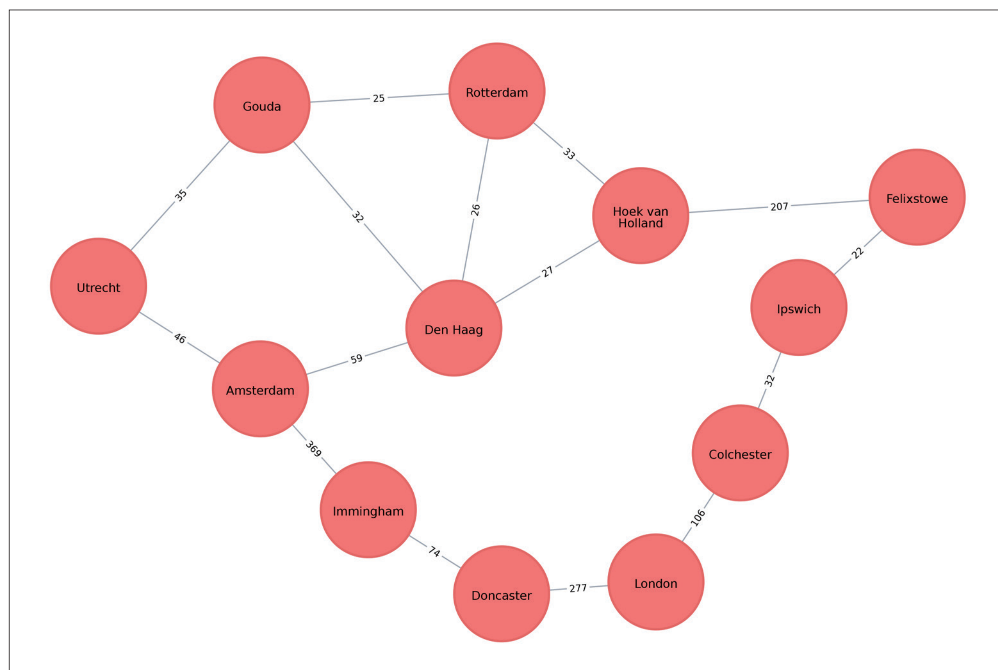


图 4-2：交通图

简单起见，将图 4-2 视为无向图，这是因为城市之间的大多数道路是双向的。由于存在少量单行道，因此如果按有向图来计算，结果会略有不同，但方法总体相似。Spark 和 Neo4j 都支持对有向图进行操作。当希望使用无向图时（例如双向道路的情况），可采用下述简便方法来实现。

- 对于 Spark，为 transport-relationships.csv 中的每一行都创建两个关系，一个从 `dst` 到 `src`，另一个从 `src` 到 `dst`。
- 对于 Neo4j，每行仅需创建一个关系，然后在运行算法时忽略关系的方向。

了解了这些简单的建模方法后，下面将图从示例 CSV 文件加载到 Spark 和 Neo4j 中。

4.1.1　将数据导入Spark

从 Spark 和 GraphFrames 包中导入所需的包：

```
from pyspark.sql.types import *
from graphframes import *
```

下面的函数在示例 CSV 文件的基础上创建一个 GraphFrame 对象。

```
def create_transport_graph():
    node_fields = [
        StructField("id", StringType(), True),
        StructField("latitude", FloatType(), True),
        StructField("longitude", FloatType(), True),
        StructField("population", IntegerType(), True)
    ]
    nodes = spark.read.csv("data/transport-nodes.csv", header=True,
                           schema=StructType(node_fields))

    rels = spark.read.csv("data/transport-relationships.csv", header=True)
    reversed_rels = (rels.withColumn("newSrc", rels.dst)
                     .withColumn("newDst", rels.src)
                     .drop("dst", "src")
                     .withColumnRenamed("newSrc", "src")
                     .withColumnRenamed("newDst", "dst")
                     .select("src", "dst", "relationship", "cost"))

    relationships = rels.union(reversed_rels)

    return GraphFrame(nodes, relationships)
```

加载节点很容易，对于关系则需要做一些预处理，要将每个关系创建两次。

下面调用该函数。

```
g = create_transport_graph()
```

4.1.2　将数据导入Neo4j

下面介绍如何将数据导入 Neo4j，首先加载节点：

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data" AS base
WITH base + "transport-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (place:Place {id:row.id})
SET place.latitude = toFloat(row.latitude),
    place.longitude = toFloat(row.latitude),
    place.population = toInteger(row.population)
```

然后加载关系：

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "transport-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (origin:Place {id: row.src})
MATCH (destination:Place {id: row.dst})
MERGE (origin)-[:EROAD {distance: toInteger(row.cost)}]->(destination)
```

虽然其中存储的是有向关系，但稍后在执行算法时将忽略方向。

4.2　广度优先搜索

广度优先搜索（breadth first search，BFS）是一个基本的图遍历算法。它从选定节点出发，首先探索其一跳范围内的所有邻节点，之后访问其两跳范围内的所有邻节点，以此类推。

1959 年，Edward F. Moore 首次提出该算法，他使用该算法找到了走出迷宫的最短路径。1961 年，C. Y. Lee 将其扩展为一种线路路由算法，参见其论文“An Algorithm for Path Connections and Its Applications”。

广度优先搜索算法常用作其他面向目标算法的基础，例如最短路径算法、连通分量算法和接近中心性算法[1]等。它还可以用于查找节点间的最短路径。

图 4-3 展示了当从荷兰城市 Den Haag（海牙）开始执行广度优先搜索时，访问交通图各个节点的顺序。城市名称旁的数字代表访问该节点的顺序。

注 1：closeness centrality，也译作亲密中心性。——译者注

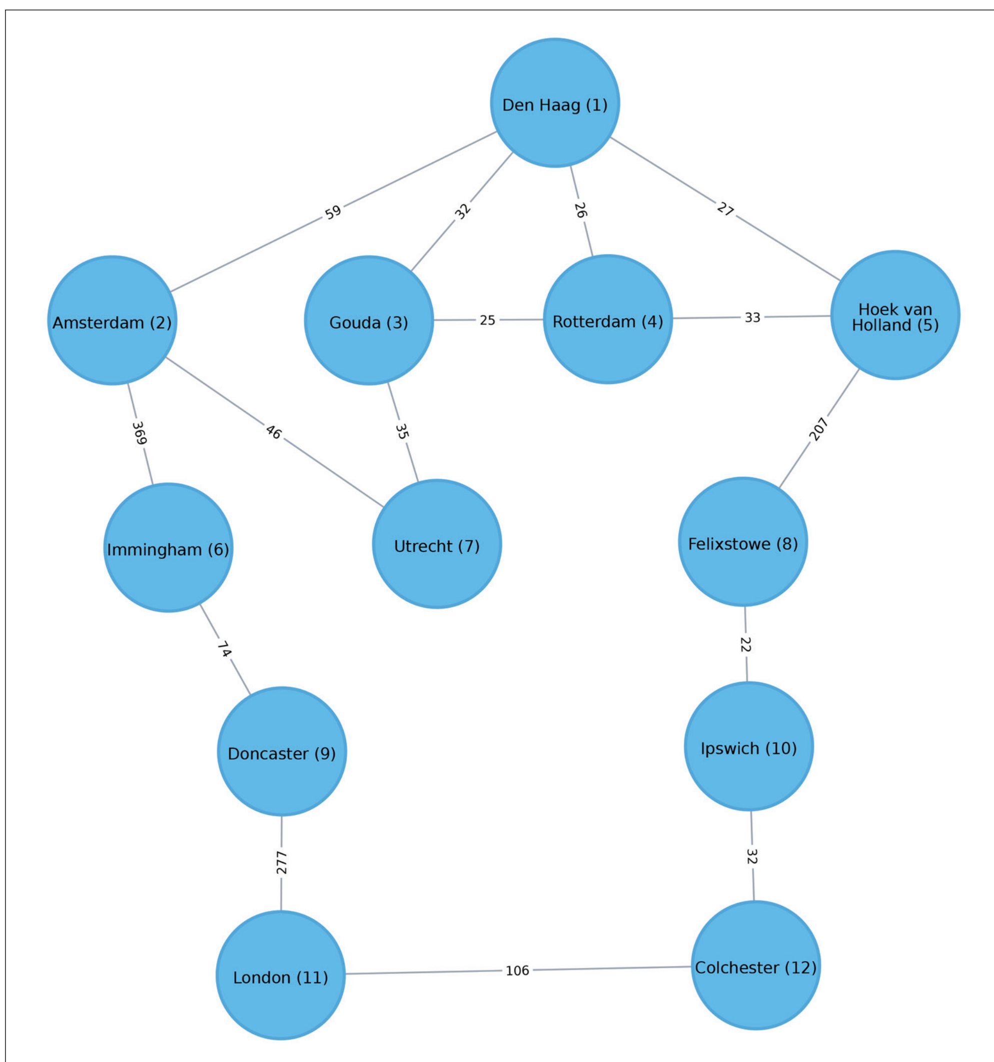


图 4-3：从 Den Haag 开始执行广度优先搜索，节点中的数值表示遍历次序

首先访问 Den Haag 的所有直接邻节点，然后访问这些节点的邻节点，再访问邻节点的邻节点，直到遍历完毕所有关系为止。

使用Spark实现广度优先搜索

在用 Spark 实现广度优先搜索算法时，可根据节点之间的关系数（跳数）来查找最短路径。可以显式命名目标节点或添加要满足的条件。

举例来说，可以使用 `bfs` 函数查找第一个人口介于 10 万和 30 万之间的中型城市（依据欧

洲标准)。先看看哪些城市的人口符合这个条件：

```
(g.vertices
 .filter("population > 100000 and population < 300000")
 .sort("population")
 .show())
```

输出结果如下所示：

id	latitude	longitude	population
Colchester	51.88921	0.90421	104390
Ipswich	52.05917	1.15545	133384

只有两个城市满足条件，我们希望基于广度优先搜索能够首先到达 Ipswich（伊普斯威奇）。

以下代码用于查找从 Den Haag 到达一个中型城市的最短路径。

```
from_expr = "id='Den Haag'"
to_expr = "population > 100000 and population < 300000 and id <> 'Den Haag'"
result = g.bfs(from_expr, to_expr)
```

`result` 对象含有描述两个城市之间各个节点和关系的列。运行以下代码来查看返回的各列。

```
print(result.columns)
```

输出结果如下：

```
['from', 'e0', 'v1', 'e1', 'v2', 'e2', 'to']
```

以 `e` 开头的列代表关系（边），而以 `v` 开头的列代表节点（顶点）。我们仅对节点感兴趣，因此从结果 DataFrame 中过滤以 `e` 开头的列。

```
columns = [column for column in result.columns if not column.startswith("e")]
result.select(columns).show()
```

在 PySpark 中运行这段代码，结果如下所示：

from	v1	v2	to
[Den Haag, 52.078...	[Hoek van Holland...	[Felixstowe, 51.9...	[Ipswich, 52.0591...

正如所愿，`bfs` 函数返回了 Ipswich。当查找到第一个匹配项时，该函数就满足条件了，如图 4-3 所示，在 Colchester 之前先计算出 Ipswich。

4.3 深度优先搜索

深度优先搜索（depth first search，DFS）是另一个基本的图遍历算法。它从选定节点出发，选择该节点的一个邻节点，然后在回溯之前尽可能沿着该路径遍历。

深度优先搜索算法最初由法国数学家 Charles Pierre Trémaux 提出，用作求解迷宫的一种策略。在场景建模中模拟可能路径时，它是一种很有用的工具。

图 4-4 展示了从 Den Haag 开始执行深度优先搜索，访问交通图中各节点的顺序。

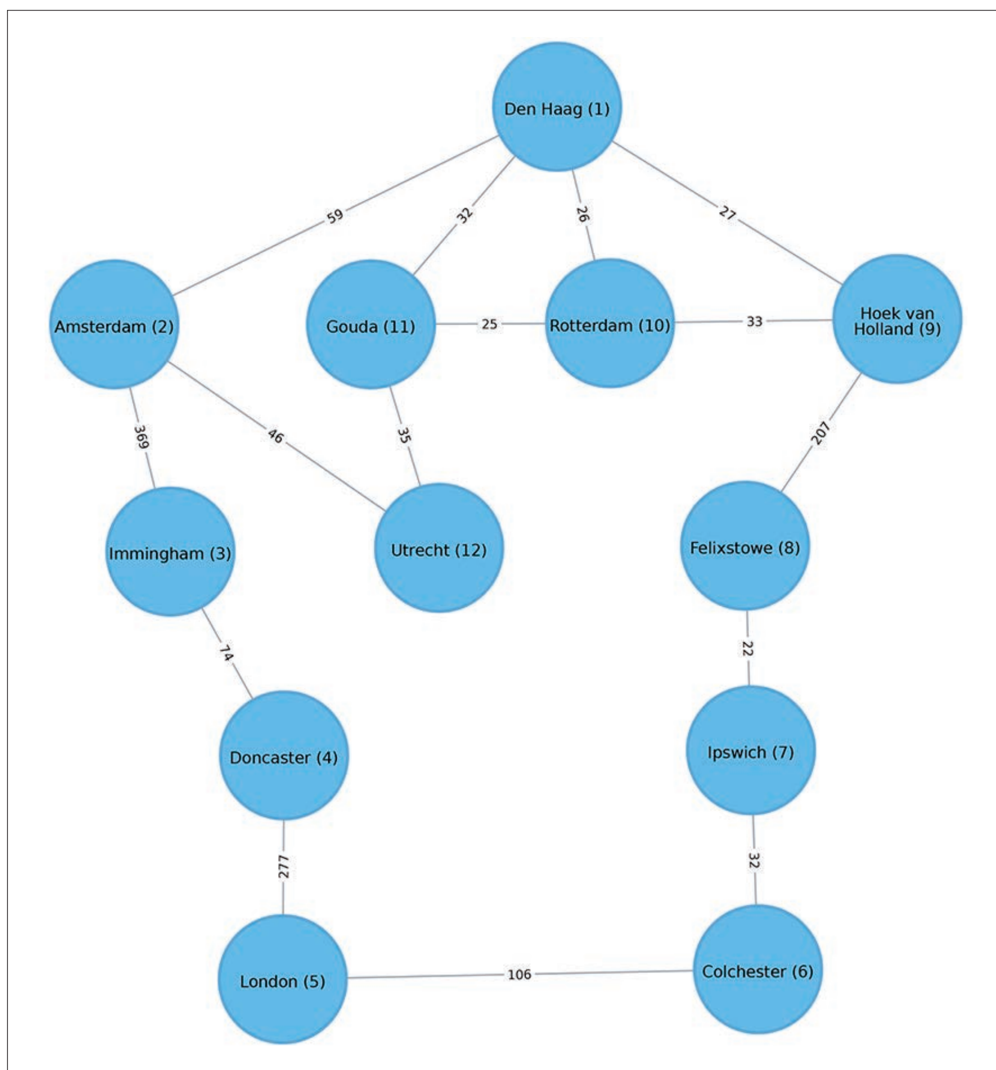


图 4-4：从 Den Haag 开始执行深度优先搜索，节点中的数值表示遍历次序

请注意，与广度优先搜索相比，节点的顺序有很大的不同。对于深度优先搜索，我们从 Den Haag 开始遍历到 Amsterdam（阿姆斯特丹），根本不需要回溯就能到达图中的其他各节点。

如前所述，搜索算法为图的遍历奠定了基础，下面看看路径查找算法，它根据跳数或权重来查找代价最小的路径。权重可以是任何可度量的东西，比如时间、距离、容量或成本等。

两种特殊的路径

图分析中有两种特殊的路径值得注意。第 1 种是**欧拉路径**，即每个关系恰好只被访问一次。第 2 种是**哈密顿路径**，即每个节点恰好只被访问一次。一条路径既可以是欧拉路径，也可以是哈密顿路径，而且如果在同一节点开始和结束，则可以称其为**环**或**回路**。图 4-5 做了直观的对比。

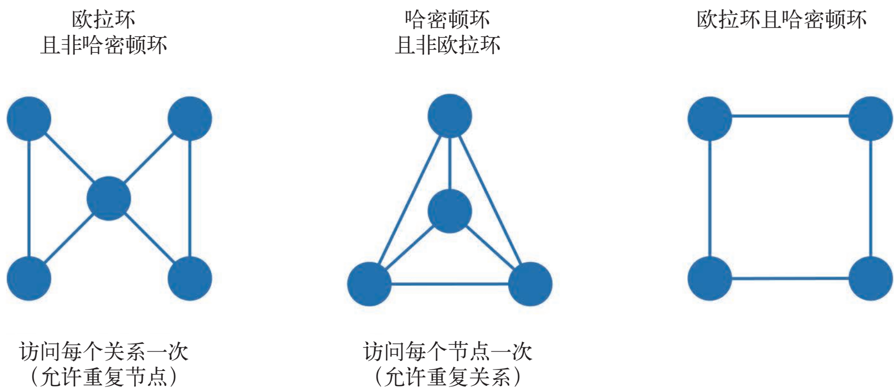


图 4-5：欧拉环和哈密顿环具有特殊的历史意义

第 1 章中的"哥尼斯堡七桥"问题实际上就是搜索一个欧拉环。很容易理解如何将其应用于引导扫雪车和邮件投递这样的线路搜索场景。然而，也有一些算法利用欧拉路径处理树结构中的数据，而且从数学角度来看要比研究其他类型的环更容易。

哈密顿环最广为人知的就是它与**旅行商问题**（traveling salesman problem，TSP）的关系，该问题是："对于旅行商而言，访问每个指定城市并返回原城市的最短路径是什么？"虽然这看起来和欧拉环类似，但是 TSP 对于近似替代方案的计算更精确。它广泛应用于各种规划、物流和优化问题中。

4.4　最短路径算法

最短路径算法用于计算一对节点之间的最短（加权）路径。由于可以实时运行，因此它对用户交互和动态工作流非常有用。

路径查找的历史可以追溯到 19 世纪，它被视为经典的图论问题。在 20 世纪 50 年代早期，它的作用凸显出来，被应用于替代路径选择，也就是当最短路径被阻塞时查找第 2 条最短

路径。1956 年，Edsger Dijkstra 提出了其中最著名的一种算法。

Edsger Dijkstra 的最短路径算法首先查找从起始节点到与其直接连接的节点的最小权重关系。该算法追踪这些权重并移动到“最近”的节点，然后针对当前节点执行同一计算，但是现在的权重是从起始节点起算的累计值。算法继续执行，计算一条累计权重“曲线”，总是选择累计权重最小的路径前进，直至到达目标节点为止。



在图分析中描述关系和路径时使用了**权重**、**代价**、**距离**和**跳**等术语。“权重”是关系的某一特定性质的数值型取值。“代价”的用法与之类似，但更常用于考察路径的总权重。

“距离”在算法中通常用于表示节点对之间的遍历代价，而且并不要求是实际的距离。“跳”通常用来表示两个节点之间的关系数量。有时会组合使用这些词，比如“到达伦敦的距离为 5 跳”或“那是这段距离的最小代价”。

4.4.1 何时使用最短路径算法

基于跳数或加权关系值，使用最短路径算法可以找到节点对之间的最优路线，例如它可以实时求解分离度以及点与点之间的最短距离或最经济的路线，也可以使用该算法直接探察特定节点之间的连通情况。

示范用例如下。

- 查找两地间的行进方向。像谷歌地图之类的互联网地图工具可以使用最短路径算法（或类似变体）提供驾驶导航服务。
- 找出社交网络中人与人之间的分离度，比如当你在 LinkedIn 上查看某人的资料时，图中会显示你们之间相隔多少人，并列出双方的共有联系人。
- 根据某个演员和 Kevin Bacon 共同出演的电影来查找他们之间的距离（**贝肯数**），在 Oracle of Bacon 网站上可以找到相关示例。Paul Erdös 是 20 世纪最杰出的数学家之一，Erdös Number 项目基于人们与 Paul Erdös 的合作关系提供了一种类似的图分析方法。



Dijkstra 算法并不支持负的权重。该算法假设向路径添加关系永远不会使路径变短——采用负的权重将违背该假设。

4.4.2 使用Neo4j实现最短路径算法

Neo4j 图算法库内置了一个程序，可用于计算无权最短路径和加权最短路径。首先介绍如何计算无权最短路径。



Neo4j 的所有最短路径算法都假定底层图是无向的。可以通过传递参数 `direction: "OUTGOING"` 或 `direction: "INCOMING"` 来重新设定。

为了让 Neo4j 的最短路径算法忽略权重，需要将 `null` 作为第 3 个参数传递给该程序，以表明在执行算法时不考虑权重，然后算法会将每个关系的默认权重设为 `1.0`：

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

该查询的返回结果如下所示：

place	cost
Amsterdam	0.0
Immingham	1.0
Doncaster	2.0
London	3.0

其中的 `cost` 值是关系（或跳）的累计总数，这与我们在 Spark 中使用广度优先搜索得到的路径相同。

还可以编写一些 Cypher 后处理代码来计算该路径的总距离。以下程序计算最短无权路径，然后得到该路径的实际代价。

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, null)
YIELD nodeId, cost

WITH collect(algo.getNodeById(nodeId)) AS path
UNWIND range(0, size(path)-1) AS index
WITH path[index] AS current, path[index+1] AS next
WITH current, next, [(current)-[r:EROAD]-(next) | r.distance][0] AS distance

WITH collect({current: current, next:next, distance: distance}) AS stops
UNWIND range(0, size(stops)-1) AS index
WITH stops[index] AS location, stops, index
RETURN location.current.id AS place,
       reduce(acc=0.0,
              distance in [stop in stops[0..index] | stop.distance] |
              acc + distance) AS cost
```

如果感觉以上代码不好理解，请注意，此间的技巧在于如何传输数据以计算整条路径的 `cost` 值。当需要累计路径代价时，牢记这一点非常有用。

该查询的返回结果如下所示：

place	cost
Amsterdam	0.0
Immingham	369.0
Doncaster	443.0
London	720.0

图 4-6 显示了从 Amsterdam 到 London 的无权最短路径，该路线通过的城市最少，总代价为 720 千米。

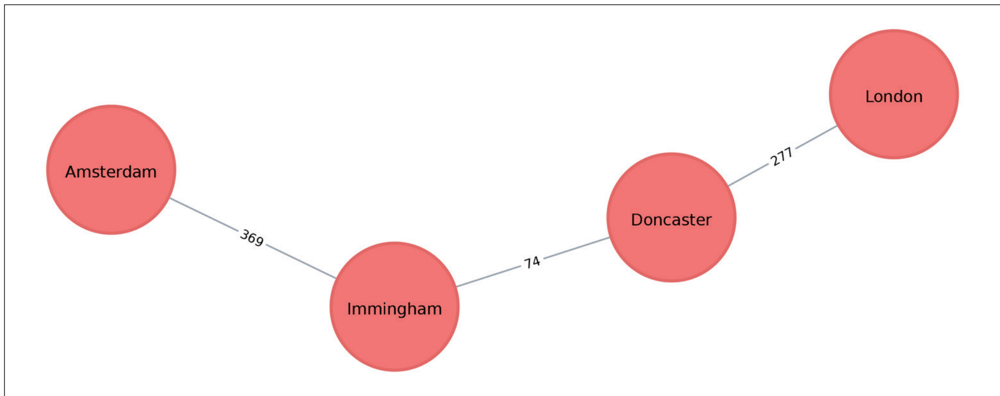


图 4-6：Amsterdam 和 London 之间的无权最短路径

对于地铁系统这样的应用场景，选择一条经过节点数最少的线路会非常有用，这是因为对乘客搭乘而言，站点数量越少越好；然而在导航应用场景中，我们可能更关心最短加权路径的总代价。

4.4.3 使用Neo4j实现加权最短路径算法

可以执行加权最短路径算法查找 Amsterdam 和 London 之间的最短路径，如下所示：

```
MATCH (source:Place {id: "Amsterdam"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.stream(source, destination, "distance")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

传递给该算法的参数有以下 3 项。

- ❑ source

  最短路径搜索的起始节点。

- ❑ `destination`

  最短路径的终止节点。

- ❑ `distance`

  表示节点对之间遍历代价的关系性质名称。

这里的代价是两地间的千米数。该查询的返回结果如下所示：

place	cost
Amsterdam	0.0
Den Haag	59.0
Hoek van Holland	86.0
Felixstowe	293.0
Ipswich	315.0
Colchester	347.0
London	453.0

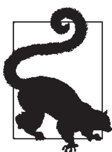
最快的路线是经过 Den Haag、Hoek van Holland（荷兰角港）、Felixstowe（费力克斯托）、Ipswich 和 Colchester（柯彻斯特）。表中的 `cost` 列给出了穿行各城市的累计代价。首先从 Amsterdam 到 Den Haag，代价是 59 千米，然后从 Den Haag 到 Hoek van Holland，累计代价为 86 千米，以此类推。最后，从 Colchester 到达 London 的总代价是 453 千米。

记住，无权重最短路径的总代价为 720 千米，而在计算最短路径时如果考虑权重，可以节省 267 千米。

4.4.4 使用Spark实现加权最短路径算法

4.2 节探讨了如何查找两个节点之间的最短路径。该最短路径是基于跳数的，这与最短**加权**路径不同，后者是指城市间总的最短距离。

如果要查找最短加权路径（在本例中是距离最短），就要用到 `cost` 性质，它可用于多种加权操作。但是该选项在 GraphFrames 中并非开箱即用，因此需要使用 `aggregateMessages` 框架编写自定义的最短加权路径算法。本书的大多数 Spark 算法示例直接调用库中的算法，但是也可以选择编写自定义函数。关于 `aggregateMessages` 的更多内容，参见 GraphFrames 用户指南的“Message passing via AggregateMessages”部分。



如果可用，建议利用已有的、经过测试的库。自己编写函数（特别是较复杂的算法），需要深入理解数据和计算原理。

应把下面的示例视为一个参考实现，在运行较大规模的数据集之前需要进行优化。如果对编写自定义函数不感兴趣，可跳过本例。

在编写自定义函数之前，先导入要用到的一些库：

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

AggregateMessages 模块是 GraphFrames 库的一部分，其中包含一些很有用的辅助函数。

下面编写自定义函数。首先创建一个用户自定义函数，用它构建源节点和目标节点之间的路径：

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```

然后定义主函数，用它计算从源节点开始的最短路径，当访问到目标节点时就立即返回。

```
def shortest_path(g, origin, destination, column_name="cost"):
    if g.vertices.filter(g.vertices.id == destination).count() == 0:
        return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
                .withColumn("path", F.array()))

    vertices = (g.vertices.withColumn("visited", F.lit(False))
                .withColumn("distance", F.when(g.vertices["id"] == origin, 0)
                            .otherwise(float("inf")))
                .withColumn("path", F.array()))
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)

    while g2.vertices.filter('visited == False').first():
        current_node_id = g2.vertices.filter('visited == False').sort
                                            ("distance").first().id

        msg_distance = AM.edge[column_name] + AM.src['distance']
        msg_path = add_path_udf(AM.src["path"], AM.src["id"])
        msg_for_dst = F.when(AM.src['id'] == current_node_id,
                             F.struct(msg_distance, msg_path))
        new_distances = g2.aggregateMessages(F.min(AM.msg).alias("aggMess"),
                                             sendToDst=msg_for_dst)

        new_visited_col = F.when(
            g2.vertices.visited | (g2.vertices.id == current_node_id),
                                                  True).otherwise(False)
        new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
                                  (new_distances.aggMess["col1"]
                                  < g2.vertices.distance),
                                  new_distances.aggMess["col1"])
                                  .otherwise(g2.vertices.distance)
        new_path_col = F.when(new_distances["aggMess"].isNotNull() &
                       (new_distances.aggMess["col1"]
                       < g2.vertices.distance), new_distances.aggMess["col2"]
                       .cast("array<string>")).otherwise(g2.vertices.path)

        new_vertices = (g2.vertices.join(new_distances, on="id",
                                         how="left_outer")
```

```
                            .drop(new_distances["id"])
                            .withColumn("visited", new_visited_col)
                            .withColumn("newDistance", new_distance_col)
                            .withColumn("newPath", new_path_col)
                            .drop("aggMess", "distance", "path")
                            .withColumnRenamed('newDistance', 'distance')
                            .withColumnRenamed('newPath', 'path'))
        cached_new_vertices = AM.getCachedDataFrame(new_vertices)
        g2 = GraphFrame(cached_new_vertices, g2.edges)
        if g2.vertices.filter(g2.vertices.id == destination).first().visited:
            return (g2.vertices.filter(g2.vertices.id == destination)
                    .withColumn("newPath", add_path_udf("path", "id"))
                    .drop("visited", "path")
                    .withColumnRenamed("newPath", "path"))
    return (spark.createDataFrame(sc.emptyRDD(), g.vertices.schema)
            .withColumn("path", F.array()))
```



如果在函数中保存了对 DataFrame 对象的引用，就需要使用 AM.getCachedDataFrame 函数缓存它们，否则在执行过程中会出现内存泄漏。在 shortest_path 函数中，使用该函数缓存 vertices 和 new_vertices 这两个 DataFrame 对象。

如果想查找 Amsterdam 和 Colchester 之间的最短路径，可按如下方式调用该函数：

```
result = shortest_path(g, "Amsterdam", "Colchester", "cost")
result.select("id", "distance", "path").show(truncate=False)
```

返回结果如下所示：

id	distance	path
Colchester	347.0	[Amsterdam, Den Haag, Hoek van Holland, Felixstowe, Ipswich, Colchester]

从 Amsterdam 到 Colchester 的最短路径全长 347 千米，途经 Den Haag、Hoek van Holland、Felixstowe 和 Ipswich。相比之下，使用深度优先搜索算法计算出的最短路径（按地点之间的关系数）经过 Immingham、Doncaster 和 London（见图 4-4）。

4.4.5　最短路径算法的变体：A*算法

A* 最短路径算法在 Dijkstra 算法的基础上进行了改进，查找最短路径的速度更快。为了便于确定下一步要探索的路径，该算法允许在启发式函数中附加额外信息。

该算法由 Peter Hart、Nils Nilsson 和 Bertram Raphael 提出，他们于 1968 年发表论文“A Formal Basis for the Heuristic Determination of Minimum Cost Paths”并在其中阐述了该算法。

在运行过程中，A* 算法在主循环的每次迭代中都会判定要扩展哪些路径。通过估计到达目标节点剩余路径的（启发式）代价，算法可以实现这一过程。



对估计路径代价采用的启发式方法要深入思考。低估路径的代价虽然会将一些本可以消除的路径包含进来，但是结果仍然准确；然而，如果在启发式方法中高估路径代价，就可能跳过本应该计算的实际较短的路径（错误地估计为较长路径），进而导致结果不准确。

A* 算法的路径选择就是要使下述函数最小化：

```
f(n) = g(n) + h(n)
```

其中，

- g(n) 是从起始节点到节点 n 的路径代价；
- h(n) 是从节点 n 到目标节点的路径代价估计，通过启发式函数计算。



在 Neo4j 实现中，采用地理空间距离进行启发式计算。在示例交通数据集中，启发式函数用到了每个地点的经纬度。

使用 Neo4j 实现 A* 算法

下述查询执行 A* 算法查找 Den Haag 和 London 之间的最短路径。

```
MATCH (source:Place {id: "Den Haag"}),
      (destination:Place {id: "London"})
CALL algo.shortestPath.astar.stream(source,
        destination, "distance", "latitude", "longitude")
YIELD nodeId, cost
RETURN algo.getNodeById(nodeId).id AS place, cost
```

传递给该算法的参数如下。

❑ source

最短路径搜索的起始节点。

❑ destination

最短路径搜索的终止节点。

❑ distance

表示遍历节点对之间路径代价的关系性质名称。这里的代价是指两个地点之间的千米数。

❑ latitude

表示每个节点纬度的节点性质名称，在地理空间启发式计算中使用。

❑ longitude

表示每个节点经度的节点性质名称，在地理空间启发式计算中使用。

运行该程序，结果如下所示：

place	cost
Den Haag	0.0
Hoek van Holland	27.0
Felixstowe	234.0
Ipswich	256.0
Colchester	288.0
London	394.0

结果与使用最短路径算法的相同，但对于更复杂的数据集而言，由于需要计算的路径更少，因此 A* 算法速度更快。

4.4.6 最短路径算法的变体：Yen的*k*最短路径算法

Yen 的 *k* 最短路径算法（简称 Yen 算法）与最短路径算法类似，但是它并不仅仅查找两对节点之间的最短路径，它还计算了第二最短路径、第三最短路径等，直到最短路径的第 *k*–1 个偏差值。

Jin Y. Yen 于 1971 年提出该算法，并在论文“Finding the K Shortest Loopless Paths in a Network”中做了系统阐述。当查找绝对最短路径并非唯一目标时，用该算法可获取替代路径。当需要一个以上的备选计划时，该算法非常有用。

使用 Neo4j 实现 Yen 算法

下面的查询执行 Yen 算法来查找 Gouda（高达）和 Felixstowe 之间的最短路径。

```
MATCH (start:Place {id:"Gouda"}),
      (end:Place {id:"Felixstowe"})
CALL algo.kShortestPaths.stream(start, end, 5, "distance")
YIELD index, nodeIds, path, costs
RETURN index,
       [node in algo.getNodesById(nodeIds[1..-1]) | node.id] AS via,
       reduce(acc=0.0, cost in costs | acc + cost) AS totalCost
```

传递给该算法的参数如下。

- start

  最短路径搜索的起始节点。

- end

  最短路径搜索的终止节点。

- 5

  最多要查找的最短路径数目。

- distance

 表示遍历节点对之间路径代价的关系性质名称，这里的代价是指两地间的千米数。

获得最短路径后，查找每个节点 ID 所对应的节点，然后从节点集合中筛选出起始节点和终止节点。

运行该程序，结果如下所示：

index	via	totalCost
0	[Rotterdam, Hoek van Holland]	265.0
1	[Den Haag, Hoek van Holland]	266.0
2	[Rotterdam, Den Haag, Hoek van Holland]	285.0
3	[Den Haag, Rotterdam, Hoek van Holland]	298.0
4	[Utrecht, Amsterdam, Den Haag, Hoek van Holland]	374.0

图 4-7 展示了 Gouda 和 Felixstowe 之间的最短路径。

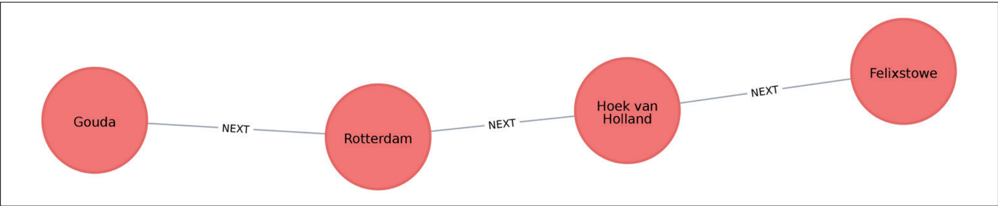


图 4-7：Gouda 和 Felixstowe 之间的最短路径

对比总代价排序结果会发现图 4-7 中的最短路径很有趣。它表明，有时可能需要考虑多条最短路径或其他参数。在本例中，第二短的路线只比最短的路线长 1 千米。如果更喜欢游览风景，可能会选择稍长一点的路线。

4.5 所有点对最短路径算法

所有点对最短路径算法（all pairs shortest path algorithm，APSP algorithm）计算所有节点对之间的最短（加权）路径。它比对图中每对节点分别运行单源最短路径算法更高效。

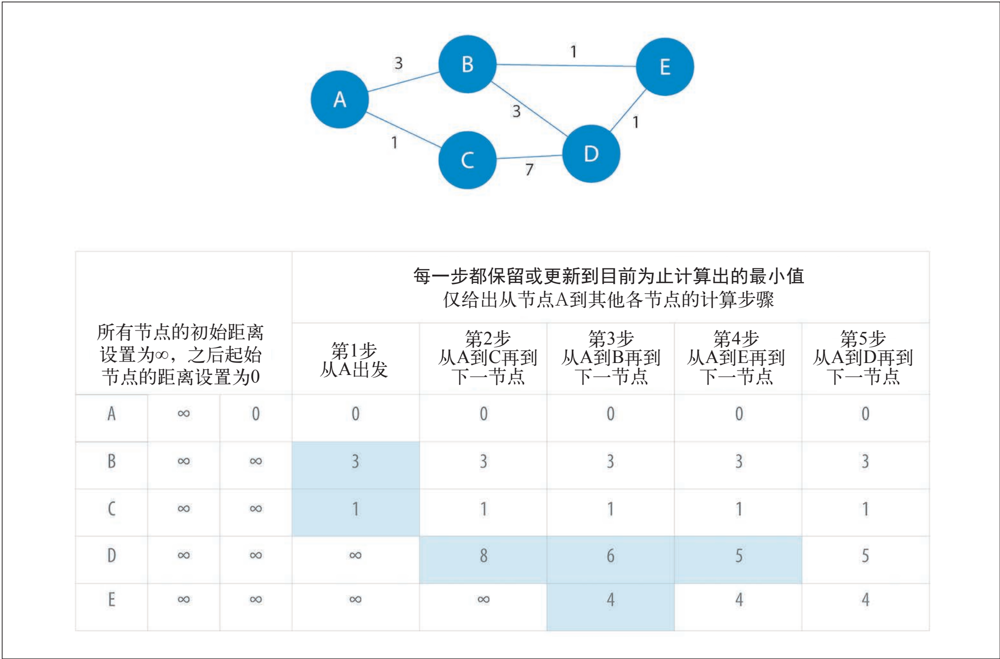
所有点对最短路径算法进一步优化了运算过程，它持续追踪距离的计算并且并行处理各节点。当计算当前节点到未经过节点的最短路径时，可以复用这些已知距离。可以结合下一节的示例更好地理解该算法的工作原理。



有些节点对之间可能彼此无法访问，这意味着这些节点之间没有最短路径。该算法不返回这些节点对间的距离。

4.5.1 近观所有点对最短路径算法

按照运算步骤的顺序很容易理解所有点对最短路径算法的计算过程。图 4-8 演示了节点 A 的处理步骤。



所有节点的初始距离设置为∞，之后起始节点的距离设置为0			每一步都保留或更新到目前为止计算出的最小值 仅给出从节点A到其他各节点的计算步骤				
			第1步 从A出发	第2步 从A到C再到下一节点	第3步 从A到B再到下一节点	第4步 从A到E再到下一节点	第5步 从A到D再到下一节点
A	∞	0	0	0	0	0	0
B	∞	∞	3	3	3	3	3
C	∞	∞	1	1	1	1	1
D	∞	∞	∞	8	6	5	5
E	∞	∞	∞	∞	4	4	4

图 4-8：计算从节点 A 到其他各节点最短路径的步骤，更新步骤用阴影标出

该算法最初假定到所有节点的距离都为无穷大。选定起始节点后，将到该节点的距离设为 0。计算过程如下。

1. 从起始节点 A 开始，计算移动到可达节点的代价并更新值。为了寻找最小值，可以选择 B（代价为 3）或 C（代价为 1）。选定 C 作为下一阶段遍历的起始节点。
2. 现在从节点 C 开始，算法更新从 A 到那些直接由 C 可达的节点的累计距离。只有当代价更小时，才会更新值。

```
A=0, B=3, C=1, D=8, E=∞
```

3. 然后选择 B 作为未访问的下一最近节点。它与节点 A、D 和 E 都有关系。算法将 A 到 B 的距离与 B 到每个节点的距离相加，计算出 A 到这些节点的距离。注意，从起始节点 A 到当前节点的最小代价始终作为沉没成本保留。距离（d）计算结果如下：

```
d(A,A) = d(A,B) + d(B,A) = 3 + 3 = 6
d(A,D) = d(A,B) + d(B,D) = 3 + 3 = 6
d(A,E) = d(A,B) + d(B,E) = 3 + 1 = 4
```

- 在这一步中，节点 A 到 B 再到 A 的距离，即 `d(A,A)=6`，大于已经计算出的最短距离（0），因此不更新其值。
- 到节点 D（6）和 E（4）的距离小于之前计算的距离，因此更新其值。

4. 接下来选择 E。现在只有达到 D（5）的累计总代价较小，因此仅更新该项。
5. 在最终计算 D 时，没有新的最小路径权重，故不更新，且算法终止。



尽管所有点对最短路径算法相对于并行计算每个节点而言更优，但对于大规模的图来说，代价仍然高昂。如果只需要计算子类别节点之间的路径，则请考虑使用子图。

4.5.2 何时使用所有点对最短路径算法

当最短路径被阻塞或不够理想时，通常采用所有点对最短路径遴选替代路线，例如使用该算法规划逻辑路线，可确保在各种路线选择情况下都有多条最佳路径。当需要考虑所有或大部分节点之间的全部可能路径时，请使用所有点对最短路径算法。

示范用例如下。

- 优化城市设施配备和商品配送，例如确定交通网格不同区段的预期交通负载。更多内容请参阅 R. C. Larson 和 A. R. Odoni 的著作“Urban Operations Research”。
- 可作为数据中心设计算法的一部分，用于查找带宽最大、延迟最低的网络。对于该方法的详细介绍，参见 A. R. Curtis 等人的论文“REWIRE: An Optimization-Based Framework for Data Center Network Design”。

4.5.3 使用Spark实现所有点对最短路径算法

Spark 的 `shortestPaths` 函数用于查找从所有节点到**路标**节点集合的最短路径。如果要查找从各个地点到 Colchester、Immingham 和 Hoek van Holland 的最短路径，可以编写以下查询：

```
result = g.shortestPaths(["Colchester", "Immingham", "Hoek van Holland"])
result.sort(["id"]).select("id", "distances").show(truncate=False)
```

在 PySpark 中运行以上代码，输出结果如下所示：

id	distances
Amsterdam	[Immingham → 1, Hoek van Holland → 2, Colchester → 4]
Colchester	[Colchester → 0, Hoek van Holland → 3, Immingham → 3]
Den Haag	[Hoek van Holland → 1, Immingham → 2, Colchester → 4]
Doncaster	[Immingham → 1, Colchester → 2, Hoek van Holland → 4]
Felixstowe	[Hoek van Holland → 1, Colchester → 2, Immingham → 4]
Gouda	[Hoek van Holland → 2, Immingham → 3, Colchester → 5]
Hoek van Holland	[Hoek van Holland → 0, Immingham → 3, Colchester → 3]

Immingham	[Immingham → 0, Colchester → 3, Hoek van Holland → 3]
Ipswich	[Colchester → 1, Hoek van Holland → 2, Immingham → 4]
London	[Colchester → 1, Immingham → 2, Hoek van Holland → 4]
Rotterdam	[Hoek van Holland → 1, Immingham → 3, Colchester → 4]
Utrecht	[Immingham → 2, Hoek van Holland → 3, Colchester → 5]

`distances` 列中每个地点旁边的数字表示从源节点到目的地所需穿越的城市之间的关系（道路）数量。在本例中，Colchester 是目的地之一，从它出发要穿越 0 个节点，但是从 Immingham 和 Hoek van Holland 出发就要经过 3 跳。在规划旅行路线时，利用这些信息有助于我们在目的地尽情畅游。

4.5.4 使用Neo4j实现所有点对最短路径算法

Neo4j 提供了所有点对最短路径算法的并行实现，将返回每对节点之间的距离。

该程序的第 1 个参数用于计算最短加权路径的性质。如果将其设为 `null`，那么算法将计算所有节点对之间的无权最短路径。

查询实现如下：

```
CALL algo.allShortestPaths.stream(null)
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).id AS source,
       algo.getNodeById(targetNodeId).id AS target,
       distance
ORDER BY distance DESC
LIMIT 10
```

该算法将返回每对节点之间的最短路径两次，每次将其中一个节点作为源节点。这在计算由单行道构成的有向图时很有用。然而，这里并不需要每条路径出现两次，所以使用谓词 `sourceNodeId < targetNodeId` 筛选结果，只保留其中一个结果即可。

查询的返回结果如下所示：

source	target	distance
Colchester	Utrecht	5.0
London	Rotterdam	5.0
London	Gouda	5.0
Ipswich	Utrecht	5.0
Colchester	Gouda	5.0
Colchester	Den Haag	4.0
London	Utrecht	4.0
London	Den Haag	4.0
Colchester	Amsterdam	4.0
Ipswich	Gouda	4.0

由于要求结果按降序排列（DESC），因此上述输出结果给出了关系最多的 10 对地点。

如果要计算最短加权路径，就不能将 null 作为第 1 个参数，可以传递含有最短路径计算所需代价值的性质名称（distance）。利用该性质可以计算每对节点之间的最短加权路径。

下面的查询可实现上述操作：

```
CALL algo.allShortestPaths.stream("distance")
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).id AS source,
       algo.getNodeById(targetNodeId).id AS target,
       distance
ORDER BY distance DESC
LIMIT 10
```

该查询的返回结果如下所示：

source	target	distance
Doncaster	Hoek van Holland	529.0
Rotterdam	Doncaster	528.0
Gouda	Doncaster	524.0
Felixstowe	Immingham	511.0
Den Haag	Doncaster	502.0
Ipswich	Immingham	489.0
Utrecht	Doncaster	489.0
London	Utrecht	460.0
Colchester	Immingham	457.0
Immingham	Hoek van Holland	455.0

这就得到了 10 对距离最远的地点。请注意，Doncaster 经常与荷兰的几个城市一起出现。如果要驾车游览这些地区，恐怕在路上的时间会很长。

4.6 单源最短路径算法

单源最短路径算法（single source shortest path algorithm，SSSP algorithm）与 Dijkstra 最短路径算法几乎同时问世，它是解决前述两个问题[2]的另一种实现。

单源最短路径算法计算从根节点到图中其他各节点的最短（加权）路径，如图 4-9 所示。

注 2：此处应该是指最短路径查找和替代路线选择这两个问题。——译者注

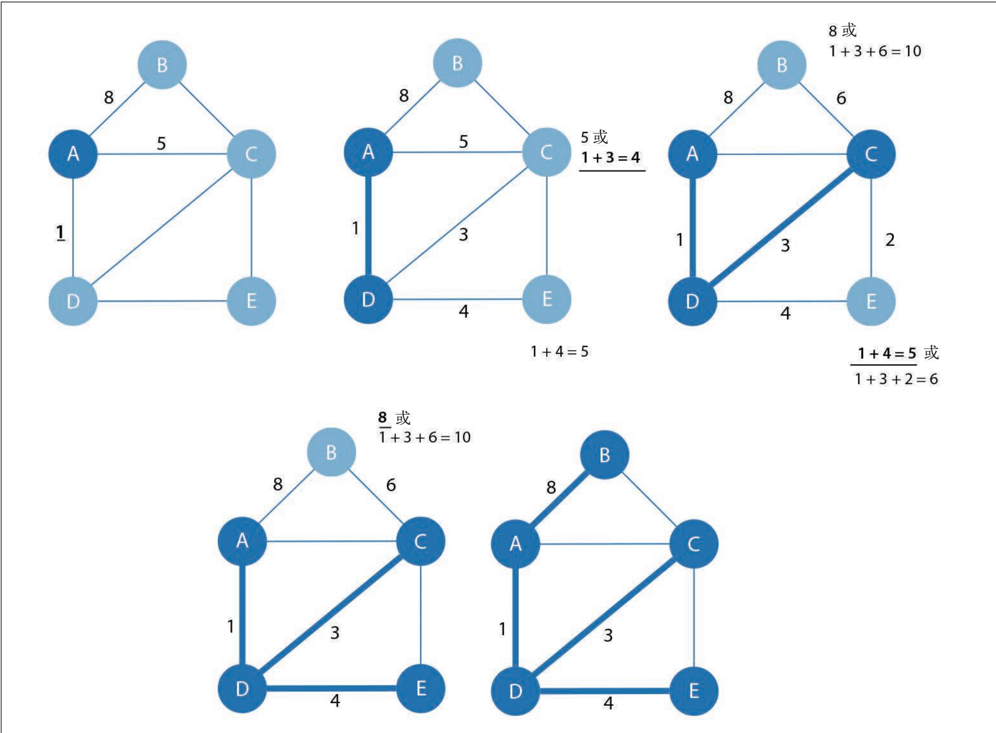


图 4-9：单源最短路径算法的步骤

算法流程如下。

1. 算法从根节点开始，且所有路径都将从根节点开始度量。在图 4-9 中，我们选择节点 A 作为根节点。
2. 选择到根节点权重最小的关系，将该关系和与其关联的节点一起添加到树中，在本例中就是 `d(A,D)=1`。
3. 选择从根节点到各未访问节点中累计权重最小的下一个关系，以相同方式将该关系和关联节点添加到树中。图 4-9 中的选项有 `d(A,B)=8`、直接路线 `d(A,C)=5`（按照 A-D-C 路线，则权重为 4）和 `d(A,E)=5`。因此，选择 A-D-C 路线，并将 C 添加到树中。
4. 持续执行该过程，直到没有更多节点要添加为止，从而得到单源最短路径。

4.6.1 何时使用单源最短路径算法

当需要求解从固定起始节点到其他各节点的最优路径时，可以采用单源最短路径算法。由于该算法基于从根节点出发的路径总权重来选择路线，因此对查找根节点到各节点的最优路径很有用，但是如果要在一次行程中访问所有节点，就不必采用该算法。

举例来说，单源最短路径算法对于确定急救服务的主要路线很有帮助，这种场景并不要求

经过每起事故的每个地点；但是该算法无法为垃圾收集这样的应用场景查找单源路线，因为这种场景需要在一次行程中访问每家每户。（对于后一种情况，要使用最小生成树算法，稍后介绍。）

示范用例如下。

- 探测拓扑变化并快速推荐新的路线，例如链接故障。
- 使用 Dijkstra 算法作为自治系统（例如局域网）中的 IP 路由选择协议。

4.6.2 使用Spark实现单源最短路径算法

我们可以采用自己编写的 `shortest_path` 函数来计算两地间的最短路径，而不必返回某一地点到其他各个地点的全部最短路径。注意，可使用 Spark 的 `aggregateMessages` 框架来自定义函数。

首先导入与之前相同的库：

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
```

使用同样的用户自定义函数来构建路径：

```
add_path_udf = F.udf(lambda path, id: path + [id], ArrayType(StringType()))
```


然后定义主函数，它用于计算从起点出发的最短路径：

```
def sssp(g, origin, column_name="cost"):
    vertices = g.vertices \
        .withColumn("visited", F.lit(False)) \
        .withColumn("distance",
            F.when(g.vertices["id"] == origin, 0).otherwise(float("inf"))) \
        .withColumn("path", F.array())
    cached_vertices = AM.getCachedDataFrame(vertices)
    g2 = GraphFrame(cached_vertices, g.edges)

    while g2.vertices.filter('visited == False').first():
        current_node_id = g2.vertices.filter('visited == False')
                            .sort("distance").first().id
        msg_distance = AM.edge[column_name] + AM.src['distance']
        msg_path = add_path_udf(AM.src["path"], AM.src["id"])
        msg_for_dst = F.when(AM.src['id'] == current_node_id,
                      F.struct(msg_distance, msg_path))
        new_distances = g2.aggregateMessages(
            F.min(AM.msg).alias("aggMess"), sendToDst=msg_for_dst)

        new_visited_col = F.when(
            g2.vertices.visited | (g2.vertices.id == current_node_id),
                                  True).otherwise(False)
        new_distance_col = F.when(new_distances["aggMess"].isNotNull() &
```

```
                                (new_distances.aggMess["col1"] <
                                g2.vertices.distance),
                                new_distances.aggMess["col1"]) \
                                .otherwise(g2.vertices.distance)
            new_path_col = F.when(new_distances["aggMess"].isNotNull() &
                            (new_distances.aggMess["col1"] <
                            g2.vertices.distance),
                            new_distances.aggMess["col2"]
                            .cast("array<string>")) \
                            .otherwise(g2.vertices.path)

            new_vertices = g2.vertices.join(new_distances, on="id",
                                            how="left_outer") \
                .drop(new_distances["id"]) \
                .withColumn("visited", new_visited_col) \
                .withColumn("newDistance", new_distance_col) \
                .withColumn("newPath", new_path_col) \
                .drop("aggMess", "distance", "path") \
                .withColumnRenamed('newDistance', 'distance') \
                .withColumnRenamed('newPath', 'path')
            cached_new_vertices = AM.getCachedDataFrame(new_vertices)
            g2 = GraphFrame(cached_new_vertices, g2.edges)

        return g2.vertices \
                    .withColumn("newPath", add_path_udf("path", "id")) \
                    .drop("visited", "path") \
                    .withColumnRenamed("newPath", "path")
```

如果要查找 Amsterdam 到其他各地的最短路径，可以像下面这样调用该函数：

```
via_udf = F.udf(lambda path: path[1:-1], ArrayType(StringType()))

result = sssp(g, "Amsterdam", "cost")
(result
 .withColumn("via", via_udf("path"))
 .select("id", "distance", "via")
 .sort("distance")
 .show(truncate=False))
```

还要定义另一个用户自定义函数，从获得的路径中筛选出起始节点和终止节点。运行以上代码，输出结果如下所示：

id	distance	via
Amsterdam	0.0	[]
Utrecht	46.0	[]
Den Haag	59.0	[]
Gouda	81.0	[Utrecht]
Rotterdam	85.0	[Den Haag]
Hoek van Holland	86.0	[Den Haag]
Felixstowe	293.0	[Den Haag, Hoek van Holland]
Ipswich	315.0	[Den Haag, Hoek van Holland, Felixstowe]

Colchester	347.0	[Den Haag, Hoek van Holland, Felixstowe, Ipswich]
Immingham	369.0	[]
Doncaster	443.0	[Immingham]
London	453.0	[Den Haag, Hoek van Holland, Felixstowe, Ipswich, Colchester]

结果列出了从根节点 Amsterdam 到图中其他各城市的物理距离（以千米为单位），按最短距离排序。

4.6.3 使用Neo4j实现单源最短路径算法

Neo4j 实现了单源最短路径算法的一个变体，称为 Delta-Stepping 算法，它将 Dijkstra 算法划分为多个可并行执行的阶段。

下面的查询执行了 Delta-Stepping 算法：

```
MATCH (n:Place {id:"London"})
CALL algo.shortestPath.deltaStepping.stream(n, "distance", 1.0)
YIELD nodeId, distance
WHERE algo.isFinite(distance)
RETURN algo.getNodeById(nodeId).id AS destination, distance
ORDER BY distance
```

查询的返回结果如下所示：

destination	distance
London	0.0
Colchester	106.0
Ipswich	138.0
Felixstowe	160.0
Doncaster	277.0
Immingham	351.0
Hoek van Holland	367.0
Den Haag	394.0
Rotterdam	400.0
Gouda	425.0
Amsterdam	453.0
Utrecht	460.0

结果列出了从根节点 London 到图中其他各城市的物理距离（以千米为单位），按最短距离排序。

4.7 最小生成树算法

最小（加权）生成树算法从给定节点开始，查找其所有可达节点以及连接这些节点权重最小的关系集合。它从任意已经过节点遍历到下一未访问节点的权重最小，从而避免了出现环。

捷克科学家 Otakar Borůvka 于 1926 年首次提出最小加权生成树算法。诞生于 1957 年的 Prim 算法，是最简单、最著名的最小生成树算法。

Prim 算法与 Dijkstra 最短路径算法类似，但它并没有在每个关系结束时都最小化总路径长度，而是将每个关系的长度分别最小化。与 Dijkstra 算法不同，它允许计算带有负权重的关系。

最小生成树算法的流程如图 4-10 所示。

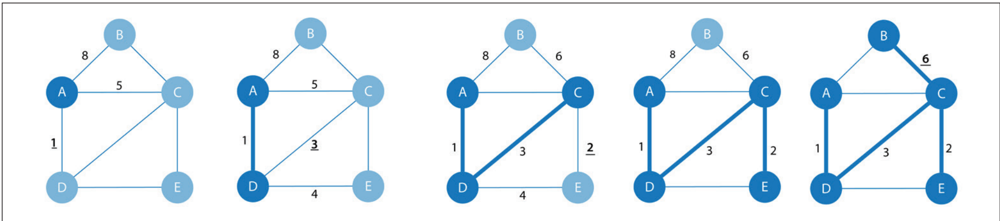


图 4-10：最小生成树算法的步骤

步骤如下。

1. 算法从只包含一个节点的树开始。在图 4-10 中，它是从节点 A 开始的。
2. 选择从该节点出发权重最小的关系并将其（连同与之关联的节点）添加到树中。在本例中为 A-D。
3. 重复此过程，始终选择权重最小且所连接节点尚不在树中的关系。如果将本例与图 4-9 中的单源最短路径算法示例进行比较，就会发现第 4 张图中的路径有所不同。这是因为单源最短路径算法基于从根节点开始的总累计值来计算最短路径，而最小生成树只考虑下一步的代价。
4. 当再没有更多节点要添加时，当前树即为最小生成树。

该算法也有许多变体，可以查找最大权重生成树（代价最大的树）和 *k* 生成树（限制树的规模）。

4.7.1 何时使用最小生成树算法

当需要查找经过所有节点的最佳路径时，请使用最小生成树。因为路径是根据每一步的代价来选择的，所以在一次行程中必须经过所有节点时，该算法非常有用。（如果不需要一条单次行程的路径，请参阅 4.6 节。）

该算法既可用于优化连通系统（如水管和电路设计）的路径，也可用于近似求解一些计算时间未知的问题，比如旅行商问题和某些类型的舍入问题。虽然不一定总能找到绝对最优解，但该算法使得潜在的复杂分析和计算密集型分析更容易实现。

示范用例如下。

- 最小化在某个国家旅行的成本。“An Application of Minimum Spanning Trees to Travel Planning”一文介绍了如何用该算法分析航空和航海的行程接续，实现旅行成本最小化。
- 可视化货币回报之间的相关性，参见论文“Minimum Spanning Tree Application in the Currency Market”。
- 追踪疫情暴发过程中的病毒感染传播历史信息，详见论文“Use of the Minimum Spanning Tree Model for Molecular Epidemiological Investigation of a Nosocomial Outbreak of Hepatitis C Virus Infection”。



最小生成树算法只有在关系权重不同的图上运行时才能得到有意义的结果。如果图没有权重，或者所有关系权重都相同，那么任何生成树都是最小生成树。

4.7.2 使用Neo4j实现最小生成树算法

接下来实际执行最小生成树算法。下面的查询找到一棵从 Amsterdam 开始的生成树：

```
MATCH (n:Place {id:"Amsterdam"})
CALL algo.spanningTree.minimum("Place", "EROAD", "distance", id(n),
  {write:true, writeProperty:"MINST"})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis, effectiveNodeCount
```

传递给算法的参数如下。

- `Place`

  在计算生成树时要考虑的节点标签。

- `EROAD`

  在计算生成树时要考虑的关系类型。

- `distance`

  表示遍历节点对之间路径代价的关系性质名称。

- `id(n)`

  生成树起始节点的内部节点 ID。

该查询在图上存储查询结果。如果想返回最小权重生成树，可以运行如下查询：

```
MATCH path = (n:Place {id:"Amsterdam"})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).id AS source, endNode(rel).id AS destination,
                                    rel.distance AS cost
```

查询的返回结果如下所示：

source	destination	cost
Amsterdam	Utrecht	46.0
Utrecht	Gouda	35.0
Gouda	Rotterdam	25.0
Rotterdam	Den Haag	26.0
Den Haag	Hoek van Holland	27.0
Hoek van Holland	Felixstowe	207.0
Felixstowe	Ipswich	22.0
Ipswich	Colchester	32.0
Colchester	London	106.0
London	Doncaster	277.0
Doncaster	Immingham	74.0

假设从 Amsterdam 出发并且希望在同一旅程中访问数据集中的其他各地点，图 4-11 展示了满足该需求的连续最短路径。

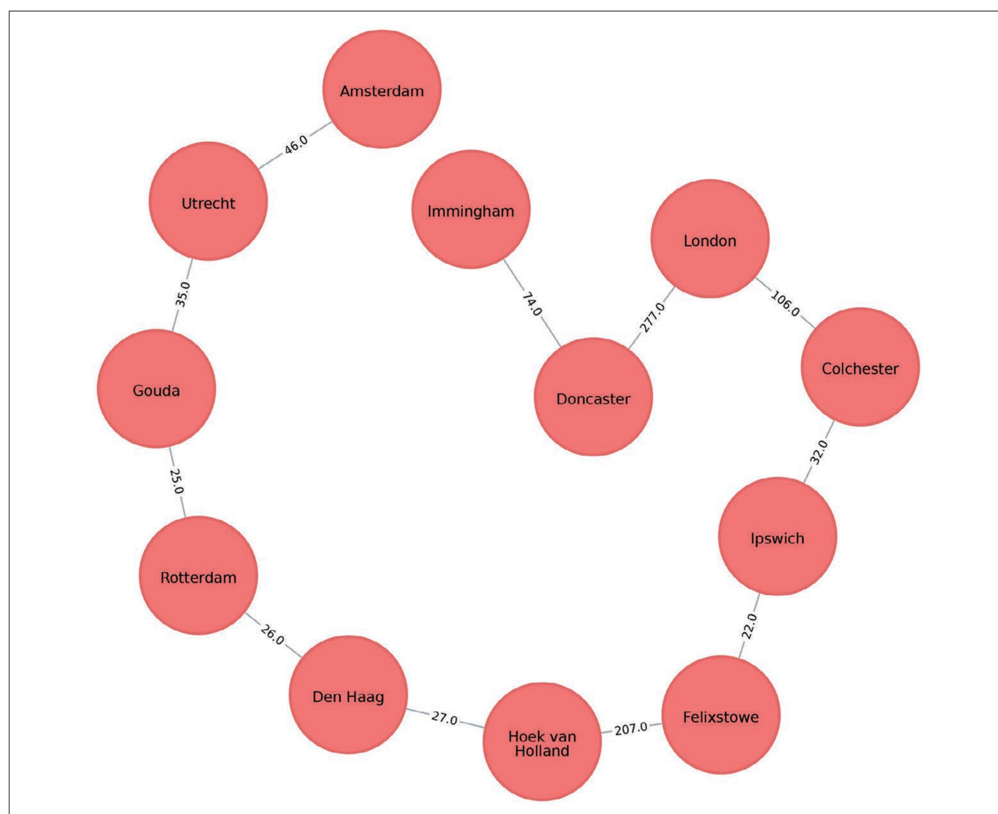


图 4-11：从 Amsterdam 出发的最小生成树

4.8 随机游走算法

随机游走算法提供了图中某条随机路径上的一个节点集合。1905 年，Karl Pearson 在给《自然》杂志的一封题为“The Problem of the Random Walk”的信中首次提到这个术语。虽然这一概念可以追溯到更早，但是直到最近，随机游走算法才被应用于网络科学中。

有时也将随机游走描述为类似于一个醉汉穿行城市。他知道自己要到达的方向或终点，但路线可能会非常迂回。

该算法从某个节点开始，带有点随机性地沿着某个关系前进——向前或向后，到达邻节点，然后从当前节点开始执行相同操作，以此类推，直到达到设定路径长度为止。（之所以说“带有点随机性”，是因为节点的关系数量及其邻节点的关系数量会影响经过该节点的概率。）

4.8.1 何时使用随机游走算法

当需要随机生成一组相互连接的节点时，可以将随机游走算法作为其他算法的一部分或数据管道使用。

示范用例如下。

- 在 node2vec 算法和 graph2vec 算法中用于创建节点嵌入。这些节点嵌入可以用作神经网络的输入。
- 在 Walktrap 和 Infomap 社团发现算法中使用。如果随机游走算法重复返回一个较小的节点集合，则表明该节点集合可能具有社团结构。
- 在机器学习模型的训练过程中使用，参见 David Mack 的论文“Review Prediction with Neo4j and TensorFlow”。

还可以阅读由 N. Masuda、M. A. Porter 和 R. Lambiotte 撰写的论文“Random Walks and Diffusion on Networks”，了解更多用例。

4.8.2 使用Neo4j实现随机游走算法

Neo4j 提供了一种随机游走算法实现。在算法的每个阶段选择下一个关系时，它支持两种模式。

- ❑ `random`

  随机选择一个关系。

- ❑ `node2vec`

  在计算前一邻节点概率分布的基础上选择关系。

下面的查询可以实现该操作：

```
MATCH (source:Place {id: "London"})
CALL algo.randomWalk.stream(id(source), 5, 1)
YIELD nodeIds
UNWIND algo.getNodesById(nodeIds) AS place
RETURN place.id AS place
```

传递给算法的参数如下。

- ❑ `id(source)`

  随机游走起始节点的内部节点 ID。

- ❑ `5`

  随机游走所用的跳数。

- ❑ `1`

  要计算的随机游走数量。

返回结果如下所示：

place
London
Doncaster
Immingham
Amsterdam
Utrecht
Amsterdam

在随机游走的每个阶段，下一关系都是随机选择的。这意味着如果重新运行算法，即使参数相同，结果也可能不同。随机游走也可能回溯到某一节点，如图 4-12 所示，从 Amsterdam 到 Utrecht 后返回。

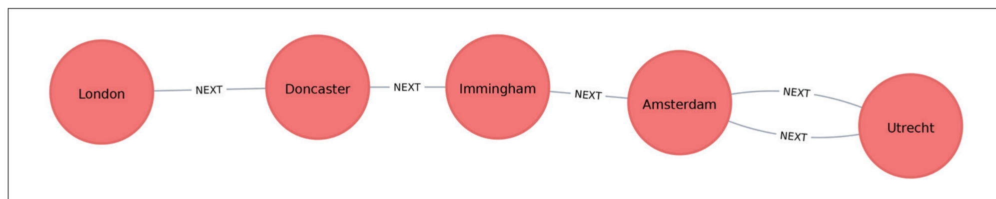


图 4-12：从 London 出发的随机游走

4.9 小结

路径查找算法有助于理解数据的关联方式。本章从基础的广度优先搜索算法和深度优先搜索算法开始，介绍了 Dijkstra 算法和其他最短路径算法，还研究了对最短路径算法优化后的一些变体，以便查找从单个节点到其他各节点的最短路径以及图中所有节点对之间的最短路径，最后还介绍了随机游走算法，它可以用于查找随机的路径集合。

接下来将介绍中心性算法，它用于查找图中有影响力的节点。

算法相关资料

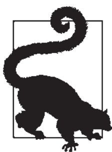
关于算法的图书有很多，其中 Steven S. Skiena 所写的《算法设计指南》尤为突出，该书阐述了图的基本概念和多种图算法。不论你想寻找有关经典算法和设计技巧的综合资料，还是只想深入研究各种算法的具体运算过程，都强烈推荐这本教科书。

第 5 章
中心性算法

中心性算法用于理解图中特定节点的作用及其对网络的影响。这些算法很有用，因为它们可以识别最重要的节点，帮助我们了解群组动态，例如可信度、可访问性、事物的传播速度以及群组之间的“桥梁”等。尽管其中许多算法是为社交网络分析而发明的，但在其他很多行业和领域中也有应用。

本章将介绍以下算法。

- 度中心性算法，可作为连通度的基准指标。
- 接近中心性算法，用于度量节点在群组中的中心程度，包括两种针对不连通群组的变体。
- 中间中心性算法[1]，用于寻找控制点，包括一种近似的替代方法。
- PageRank 算法，用于了解总体影响，包括流行的个性化选项。



度量内容不同，中心性算法产生的结果也会显著不同。如果结果不理想，就应检查所用算法是否与其设计初衷一致。

本章将解释这些算法的工作原理，并给出 Spark 示例及 Neo4j 示例。如果算法在其中一个平台上不可用或者差异不明显，则仅提供一个平台的示例。

图 5-1 展示了中心性算法所针对的各种问题之间的区别，表 5-1 是每种算法及其示范用例的速查表。

注 1：Betweenness Centrality，也译作中介中心性算法。——译者注

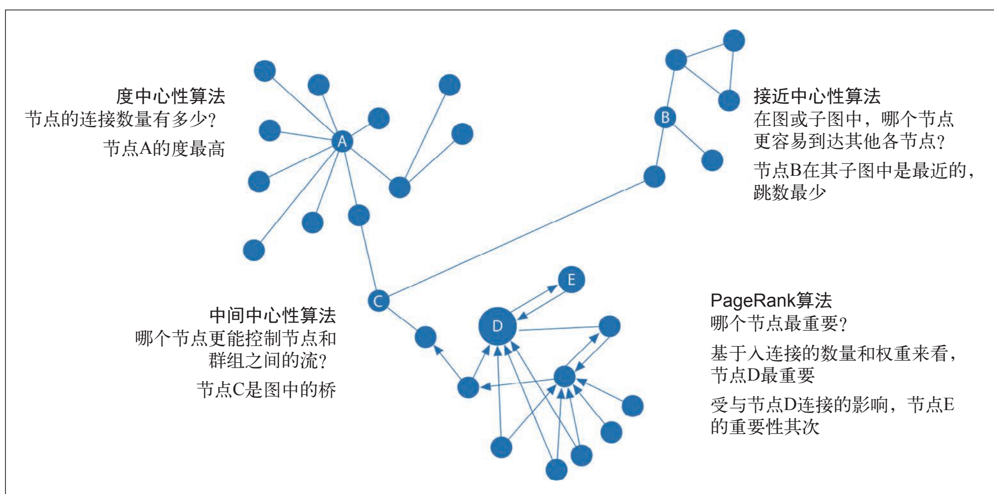


图 5-1：具有代表性的中心性算法及其针对的问题类型

表5-1：中心性算法总览

算法类型	作　　用	示范用例	Spark 示例	Neo4j 示例
度中心性算法	度量节点拥有的关系数量	通过研究入度来评估某人的受欢迎程度，使用出度评估其社交情况	有	无
接近中心性算法 变体：Wasserman & Faust 算法、调和中心性算法	计算哪些节点具有到其他各节点的最短路径	查找最佳位置，最大化新型公共服务的可达性	有	有
中间中心性算法 变体：RA-Brandes 算法	度量通过某个节点的最短路径的数量	通过查找针对特定疾病的控制基因来改进药物的靶向性	无	有
PageRank 算法 变体：个性化 PageRank 算法	通过与当前节点相连的节点以及它们的邻节点来估计当前节点的重要性（由谷歌公司推广）	为机器学习的抽取阶段查找最有影响力的特征，以及为自然语言处理中的实体关联进行文本排序	有	有



有几种中心性算法需要计算每对节点之间的最短路径。这在中小型图上可正常运行，对于大型图则无法计算。为了避免在大型图上长时间运行，有些算法（例如中间中心性算法）提供了近似版本。

首先介绍示例数据集，演示如何将数据导入 Spark 和 Neo4j，然后按表 5-1 的顺序介绍每种算法：先是概述，然后在必要时介绍算法的运算过程。对于算法的变体，讲过的内容不再赘述。很多地方还会就相关算法的使用场景给出指导。理论先行，而后用样本数据集演示示例代码。

那就开始吧！

5.1 示例数据：社交图

中心性算法可用于所有类型的图，只不过在考察动态影响和信息流方面，社交网络更容易理解。本章的示例（见表 5-2 和表 5-3）在一个类 Twitter 的小型图上运行。本书配套文件包含用于创建图的节点和关系的文件。

表5-2：social-nodes.csv

id
Alice
Bridget
Charles
Doug
Mark
Michael
David
Amy
James

表5-3：social-relationships.csv

src	dst	relationship
Alice	Bridget	FOLLOWS
Alice	Charles	FOLLOWS
Mark	Doug	FOLLOWS
Bridget	Michael	FOLLOWS
Doug	Mark	FOLLOWS
Michael	Alice	FOLLOWS
Alice	Michael	FOLLOWS
Bridget	Alice	FOLLOWS
Michael	Bridget	FOLLOWS
Charles	Doug	FOLLOWS
Bridget	Doug	FOLLOWS
Michael	Doug	FOLLOWS
Alice	Doug	FOLLOWS
Mark	Alice	FOLLOWS
David	Amy	FOLLOWS
James	David	FOLLOWS

图 5-2 展示了我们要构造的图。

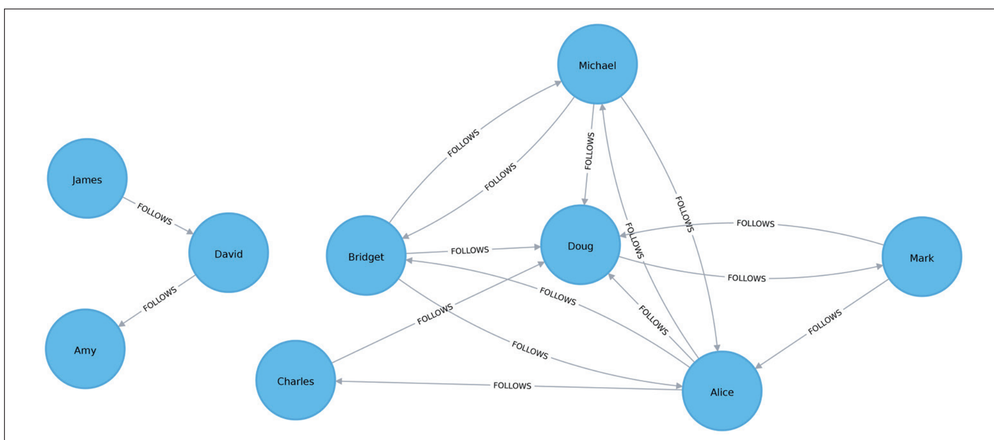


图 5-2：示例图模型

我们有一个较大的用户集（其中包含用户间的关联关系）和一个较小的用户集（它与较大的用户集没有关联）。

下面基于这些 CSV 文件中的内容在 Spark 和 Neo4j 中创建图。

5.1.1　将数据导入Spark

从 Spark 和 GraphFrames 包中导入所需的包：

```
from graphframes import *
from pyspark import SparkContext
```

利用以下代码创建一个基于 CSV 文件内容的 GraphFrame 对象。

```
v = spark.read.csv("data/social-nodes.csv", header=True)
e = spark.read.csv("data/social-relationships.csv", header=True)
g = GraphFrame(v, e)
```

5.1.2　将数据导入Neo4j

向 Neo4j 加载数据，下述查询导入了节点：

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:User {id: row.id})
```

下述查询导入了关系：

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "social-relationships.csv" AS uri
```

```
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:User {id: row.src})
MATCH (destination:User {id: row.dst})
MERGE (source)-[:FOLLOWS]->(destination)
```

图已经加载完毕，接下来介绍算法。

5.2 度中心性算法

度中心性算法是本书中最简单的算法。它计算节点的输入关系数和输出关系数，用于在图中查找受欢迎的节点。该算法是 1979 年由 Linton C. Freeman 在其论文“Centrality in Social Networks: Conceptual Clarification”中提出的。

5.2.1 可达性

了解节点的可达性是一种相当重要的度量方法。节点的**度**是其拥有的直接关系数，可按入度和出度两种指标计算。可以将其视为节点的直接可达度，例如在一个活跃的社交网络中，度较高的人会有很多直接联系人，更有可能在该网络中传播流感病毒。

网络的**平均度**就是关系总数除以节点总数得到的值，这个值可能会被度较高的节点严重扭曲。**度分布**是指随机选择的节点拥有特定关系数的概率。

图 5-3 展示了某在线论坛各主题之间实际连接分布的差异。如果简单地取平均值，就会以为大多数主题有 10 个连接，而实际上大多数主题只有两个连接。

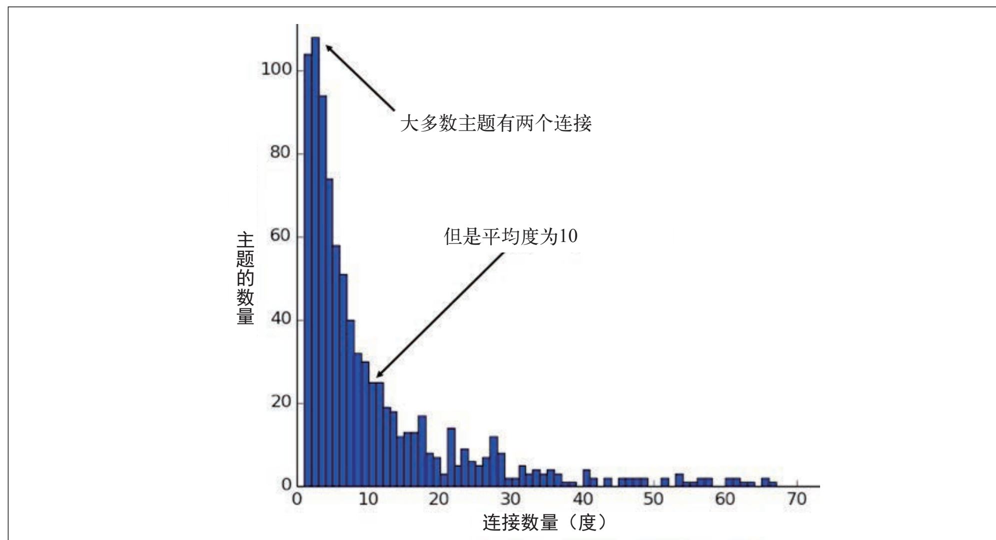


图 5-3：Jacob Silterrapa 绘制的度分布图给出了一个示例，说明平均值往往不能反映网络的实际分布（CC BY-SA 3.0）

这些指标可用于划分网络类型，如第 2 章中讨论的无标度网络或小世界网络。它们还有助于快速评估事物在网络中传播或推广的可能性。

5.2.2 何时使用度中心性算法

如果想通过研究输入关系和输出关系的数量来分析影响程度，或者想了解单个节点的“受欢迎程度”，请使用度中心性算法。当需要考虑直接连通情况或近期可能性等问题时，这种算法很有效。当要评估整个图的最小度、最大度、平均度和标准差时，度中心性算法也适用于全局分析。

示范用例如下。

- 通过关系（比如社交网络中的关联关系）来识别有影响力的个体，例如在 BrandWatch 的“2017 年 Twitter 最具影响力男性和女性”榜单中，每一类别的前 5 名的粉丝都超过 4000 万。
- 区分诈骗分子和在线拍卖网站的合法用户。由于共谋的目的是人为抬价，因此诈骗分子的加权中心度往往要高得多。请参阅 Phiradet Bangcharoensap 等人的论文“Two Step Graph-Based Semi-Supervised Learning for Online Auction Fraud Detection”。

5.2.3 使用Spark实现度中心性算法

使用以下代码执行度中心性算法：

```
total_degree = g.degrees
in_degree = g.inDegrees
out_degree = g.outDegrees

(total_degree.join(in_degree, "id", how="left")
 .join(out_degree, "id", how="left")
 .fillna(0)
 .sort("inDegree", ascending=False)
 .show())
```

首先计算总的入度和出度，然后对这些 DataFrame 对象执行连接操作，使用左连接来保留没有输入关系或输出关系的节点。如果节点没有关系，则使用 `fillna` 函数将该值设为 0。

在 PySpark 中运行以上代码，结果如下所示：

id	degree	inDegree	outDegree
Doug	6	5	1
Alice	7	3	4
Michael	5	2	3
Bridget	5	2	3
Charles	2	1	1

Mark	3	1	2
David	2	1	1
Amy	1	1	0
James	1	0	1

图 5-4 显示，Doug 是该图中最受欢迎的用户，他有 5 个粉丝（入连接）。图中这部分的所有用户都关注他，而他只关注一个人。在真实的 Twitter 网络中，名人的粉丝数量众多，但他们关注的人往往很少，因此可以认为 Doug 是名人。

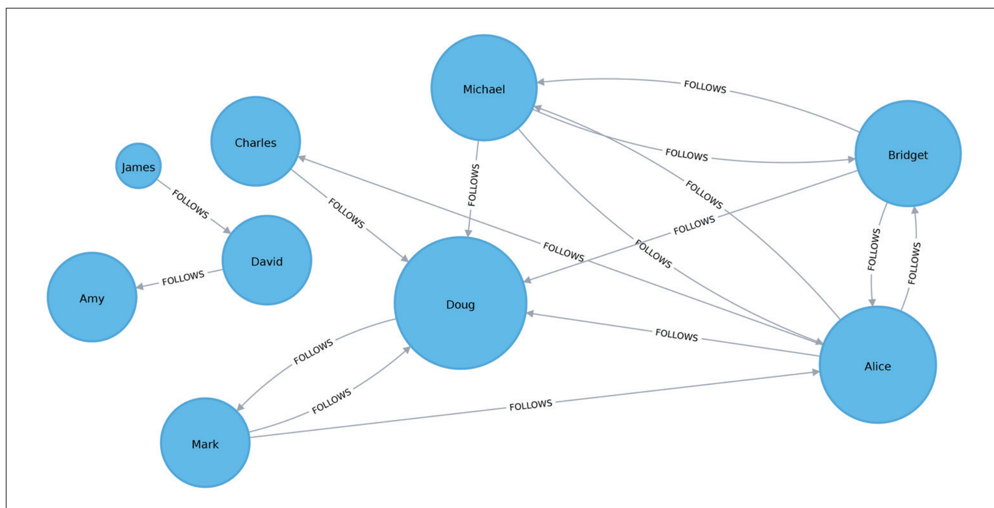


图 5-4：度中心性的可视化

如果要创建一个显示最受关注用户的网页，或者想推荐可关注的人，就可以使用该算法来识别这些人。



有些数据可能包含关系非常稠密（有大量关系）的节点。这并不会增加太多额外信息，但是会扭曲一些结果或增加计算复杂度。应该通过子图过滤这些稠密的关系，或者采用投影方法汇总关系权重。

5.3 接近中心性算法

接近中心性算法用于发现可通过子图高效传播信息的节点。

衡量节点中心性的指标是其到其他各节点的平均距离（反距离）。接近中心性得分高的节点与其他各节点的距离最短。

对于每个节点，接近中心性算法在计算所有节点对之间的最短路径的基础上，还要计算它到其他各节点的距离之和，然后对得到的和**求倒数**，以确定该节点的接近中心性得分。

节点的接近中心性计算公式为：

$$C(u) = \frac{1}{\sum_{v=1}^{n-1} d(u,v)}$$

其中：

- u 为节点；
- n 为图中节点数；
- $d(u, v)$ 是另一节点 v 和节点 u 之间的最短距离。

更常见的做法是将该得分归一化，以此表示最短路径的平均长度，而不是最短路径之和。利用这一修正方法可以比较在不同规模的图中节点的接近中心性。

接近中心性的归一化公式如下。

$$C_{\text{norm}}(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

5.3.1　何时使用接近中心性算法

当需要知道哪个节点的传播速度最快时，可以使用接近中心性算法。在评估通信和行为分析中的交互速度时，使用加权关系尤其有用。

示范用例如下。

- 发现处于有利位置的个体，以控制和获取组织内的重要信息和资源。Valdis E. Krebs 的论文“Mapping Networks of Terrorist Cells”就是这样一项研究。
- 在远程通信和数据包传递中作为估算到达时间的启发式函数，使要传递的信息通过最短路径流向预定义目标。该算法也可用于揭示通过所有最短路径同时进行传播的情形，例如通过局部社团扩散感染。更多细节请参阅 Stephen P. Borgatti 的论文“Centrality and Network Flow”。
- 以基于图的关键短语抽取过程为基础，评估单词在文档中的重要性。Florian Boudin 在其论文“A Comparison of Centrality Measures for Graph-Based Keyphrase Extraction”中阐述了该过程。



接近中心性在连通图上效果更好。当把原始公式应用于非连通图时，没有路径的两个节点之间的距离是无穷大的。这意味着如果将到该节点的所有距离相加，将得到无穷大的接近中心性得分。为了避免这个问题，后文将介绍原始公式的一个变体。

5.3.2 使用Spark实现接近中心性算法

Spark 没有内置接近中心性算法，但可以使用 `aggregateMessages` 框架来自己实现算法，4.4.4 节介绍过该框架。

在创建函数之前，先导入一些要用到的库：

```
from graphframes.lib import AggregateMessages as AM
from pyspark.sql import functions as F
from pyspark.sql.types import *
from operator import itemgetter
```

我们还将创建一些用户自定义函数，以供后续使用：

```
def collect_paths(paths):
    return F.collect_set(paths)


collect_paths_udf = F.udf(collect_paths, ArrayType(StringType()))

paths_type = ArrayType(
    StructType([StructField("id", StringType()), StructField("distance",


def flatten(ids):
    flat_list = [item for sublist in ids for item in sublist]
    return list(dict(sorted(flat_list, key=itemgetter(0))).items())


flatten_udf = F.udf(flatten, paths_type)


def new_paths(paths, id):
    paths = [{"id": col1, "distance": col2 + 1} for col1,
                            col2 in paths if col1 != id]
    paths.append({"id": id, "distance": 1})
    return paths


new_paths_udf = F.udf(new_paths, paths_type)


def merge_paths(ids, new_ids, id):
    joined_ids = ids + (new_ids if new_ids else [])
    merged_ids = [(col1, col2) for col1, col2 in joined_ids if col1 != id]
    best_ids = dict(sorted(merged_ids, key=itemgetter(1), reverse=True))
```

```
        return [{"id": col1, "distance": col2} for col1, col2 in best_ids.items()]


    merge_paths_udf = F.udf(merge_paths, paths_type)


    def calculate_closeness(ids):
        nodes = len(ids)
        total_distance = sum([col2 for col1, col2 in ids])
        return 0 if total_distance == 0 else nodes * 1.0 / total_distance


    closeness_udf = F.udf(calculate_closeness, DoubleType())
```

计算每个节点接近中心性得分的主体代码如下：

```
vertices = g.vertices.withColumn("ids", F.array())
cached_vertices = AM.getCachedDataFrame(vertices)
g2 = GraphFrame(cached_vertices, g.edges)

for i in range(0, g2.vertices.count()):
    msg_dst = new_paths_udf(AM.src["ids"], AM.src["id"])
    msg_src = new_paths_udf(AM.dst["ids"], AM.dst["id"])
    agg = g2.aggregateMessages(F.collect_set(AM.msg).alias("agg"),
                               sendToSrc=msg_src, sendToDst=msg_dst)
    res = agg.withColumn("newIds", flatten_udf("agg")).drop("agg")
    new_vertices = (g2.vertices.join(res, on="id", how="left_outer")
                    .withColumn("mergedIds", merge_paths_udf("ids", "newIds",
                    "id")).drop("ids", "newIds")
                    .withColumnRenamed("mergedIds", "ids"))
    cached_new_vertices = AM.getCachedDataFrame(new_vertices)
    g2 = GraphFrame(cached_new_vertices, g2.edges)

(g2.vertices
 .withColumn("closeness", closeness_udf("ids"))
 .sort("closeness", ascending=False)
 .show(truncate=False))
```

运行这段代码，输出结果如下所示：

id	ids	closeness
Doug	[[Charles, 1], [Mark, 1], [Alice, 1], [Bridget, 1], [Michael, 1]]	1.0
Alice	[[Charles, 1], [Mark, 1], [Bridget, 1], [Doug, 1], [Michael, 1]]	1.0
David	[[James, 1], [Amy, 1]]	1.0
Bridget	[[Charles, 2], [Mark, 2], [Alice, 1], [Doug, 1], [Michael, 1]]	0.7142857142857143
Michael	[[Charles, 2], [Mark, 2], [Alice, 1], [Doug, 1], [Bridget, 1]]	0.7142857142857143
James	[[Amy, 2], [David, 1]]	0.6666666666666666
Amy	[[James, 2], [David, 1]]	0.6666666666666666
Mark	[[Bridget, 2], [Charles, 2], [Michael, 2], [Doug, 1], [Alice, 1]]	0.625
Charles	[[Bridget, 2], [Mark, 2], [Michael, 2], [Doug, 1], [Alice, 1]]	0.625

Alice、Doug 和 David 是图中连接最密集的节点，得分为 `1.0`，这意味着每个节点都与这部分图中的所有节点直接相连。图 5-5 表明，尽管 David 只有几个关系，但在他的朋友圈中这些关系非常重要。换言之，该分值表示每个用户在其子图（而不是整个图）中与其他用户的亲密程度。

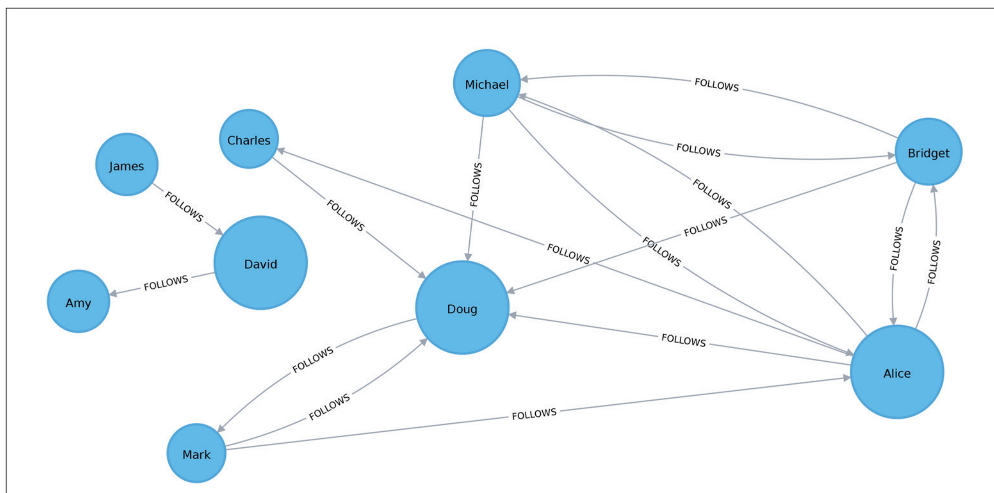


图 5-5：接近中心性的可视化

5.3.3　使用Neo4j实现接近中心性算法

Neo4j 使用下述公式实现接近中心性算法：

$$C(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}$$

其中：

- u 为节点；
- n 是与 u 位于同一分量（子图或群组）中的节点的数量；
- $d(u, v)$ 是另一个节点 v 与 u 之间的最短距离。

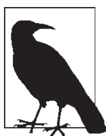
调用以下程序，计算图中每个节点的接近中心性得分：

```
CALL algo.closeness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id, centrality
ORDER BY centrality DESC
```

运行该程序，输出结果如下所示：

user	centrality
Alice	1.0
Doug	1.0
David	1.0
Bridget	0.7142857142857143
Michael	0.7142857142857143
Amy	0.6666666666666666
James	0.6666666666666666
Charles	0.625
Mark	0.625

结果与 Spark 算法实现的相同，与之前一样，分值表示他们与子图（而不是整个图）中其他用户的亲密程度。



根据接近中心性算法的严格解释，图中所有节点的得分都是∞，因为每个节点至少有一个无法到达的节点，不过按照每个分量分别计算得分通常更有用。

在理想情况下，我们希望得到整个图的接近中心性指标，因此接下来介绍接近中心性算法的一些变体。

5.3.4 接近中心性算法变体：Wasserman & Faust算法

Stanley Wasserman 和 Katherine Faust 提出了一个改进公式，用于计算包含多个非连通子图的图的接近中心性得分。他们在《社会网络分析：方法与应用》一书中详细介绍了该公式。该公式得到的结果是群组中可达节点数与到可达节点平均距离的比值。

公式如下：

$$C_{\mathrm{WF}}(u) = \frac{n-1}{N-1}\left(\frac{n-1}{\sum_{v=1}^{n-1} d(u,v)}\right)$$

其中：

- u 为节点；
- N 为总的节点数；
- n 是与 u 在同一分量中的节点的数量；
- $d(u, v)$ 是另一节点 v 到 u 的最短距离。

传递参数 `improved: true`，告诉接近中心性计算程序采用该公式。

下面的查询使用 Wasserman & Faust 公式来执行接近中心性算法：

```
CALL algo.closeness.stream("User", "FOLLOWS", {improved: true})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

结果如下所示：

user	centrality
Alice	0.5
Doug	0.5
Bridget	0.35714285714285715
Michael	0.35714285714285715
Charles	0.3125
Mark	0.3125
David	0.125
Amy	0.08333333333333333
James	0.08333333333333333

如图 5-6 所示，现在的结果更能代表整个图中各节点的亲密程度。较小子图的成员（David、Amy 和 James）的得分已经降低，现在他们在所有用户中的得分最低。这是合理的，因为他们是孤立的节点集。该公式对于检测一个节点在整个图（而不是在其子图）中的重要性更有用。

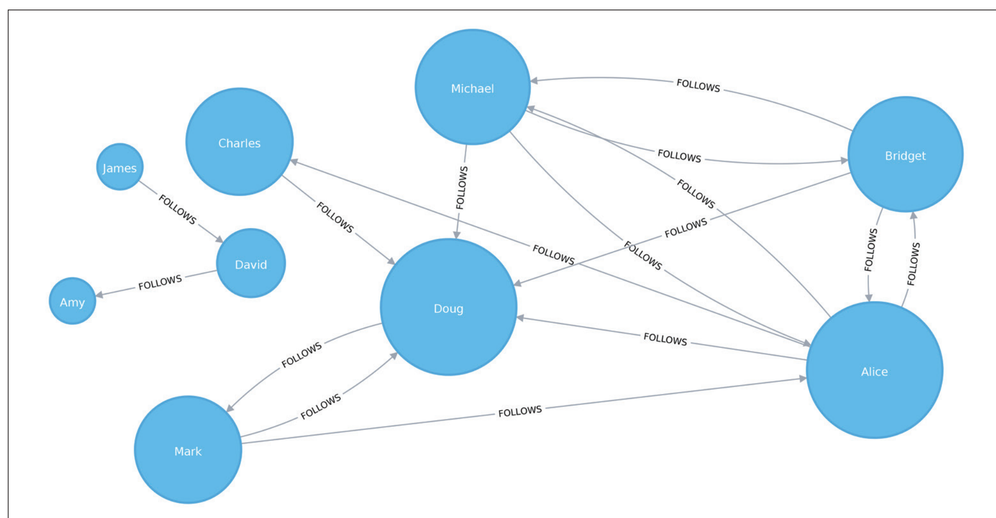


图 5-6：接近中心性的可视化

下面介绍调和中心性算法，该算法使用另一个公式计算接近中心性得分，可以得到相似的结果。

5.3.5 接近中心性算法变体：调和中心性算法

调和中心性（Harmonic Centrality，也称 Valued Centrality）算法是接近中心性算法的一种变体，是为了解决非连通图的固有问题而发明的。在“Harmony in a Small World”这篇论文中，Massimo Marchiori 和 Vito Latora 将此概念作为平均最短路径的实际表示形式。

在计算每个节点的亲密度分值时，并不是累加一个节点到其他各节点的距离，而是将这些距离的倒数相加，这意味着 ∞ 值变得无关紧要了。

原始的节点调和中心性计算公式如下：

$$H(u)=\sum_{v=1}^{n-1}\frac{1}{d(u,v)}$$

其中：

- u 是节点；
- n 是图中的节点数；
- $d(u, v)$ 是另一节点 v 到 u 的最短距离。

对于接近中心性，也可以用以下公式计算归一化调和中心性得分：

$$H_{\text{norm}}(u)=\frac{\sum_{v=1}^{n-1}\frac{1}{d(u,v)}}{n-1}$$

在该公式中，∞ 值得到妥善处理。

使用 Neo4j 实现调和中心性算法

下述查询执行调和中心性算法：

```
CALL algo.closeness.harmonic.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

运行该程序，结果如下所示：

user	centrality
Alice	0.625
Doug	0.625
Bridget	0.5
Michael	0.5
Charles	0.4375
Mark	0.4375
David	0.25
Amy	0.1875
James	0.1875

该算法的结果与原始接近中心性算法有所不同，但与改进的 Wasserman & Faust 算法的结果相似。当处理具有多个连通分量的图时，可以使用这两种算法。

5.4 中间中心性算法

在某些系统中，最关键的要素并不是拥有绝对权威或最高地位的要素。有时是中间人将各个群体联系起来，或者说，中间人对资源或信息流的控制权最大。中间中心性算法检测节点对图中信息流或资源的影响程度，通常用于查找将图的一部分与另一部分桥接的节点。

中间中心性算法计算连通图中每对节点之间的最短（加权）路径。每个节点的分值根据通过该节点的最短路径数量确定。通过节点的最短路径数量越多，其得分就越高。

1977 年，Linton C. Freeman 在其论文“A Set of Measures of Centrality Based on Betweenness”中提出中间中心性，并将其称为“关于中心性的 3 个截然不同的直观概念”之一。

5.4.1 桥与控制点

网络中的桥可以是节点或关系。在非常简单的图中，可以通过查找节点或关系来寻找桥。如果删除某一节点或关系将导致图的某个部分不连通，那么该节点或关系就是桥。然而，这种方法在典型的图中并不实用，我们采用中间中心性算法。还可以将群组视为节点，借此度量簇的中间中心性。

如果一个节点位于另外两个节点之间的每一条最短路径上，则该节点是这两个节点的**中枢节点**，如图 5-7 所示。

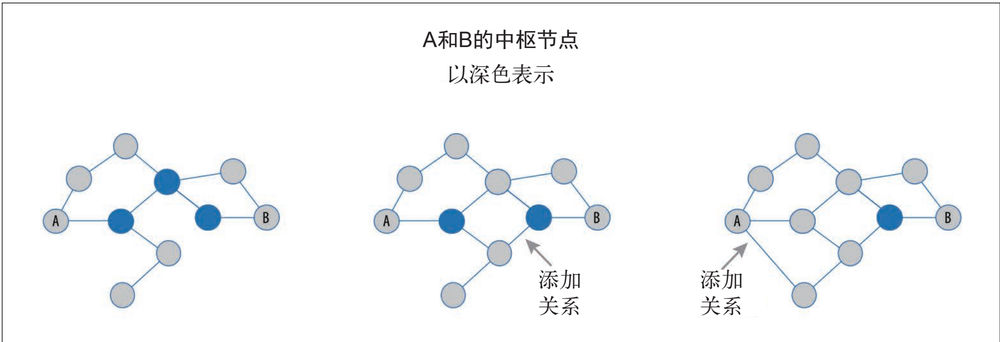


图 5-7：中枢节点位于两个节点之间的每条最短路径上。创建更多最短路径可以减少中枢节点的数量，可用于降低风险等方面

中枢节点在连接其他节点方面发挥着重要作用——如果删除一个中枢节点，则原节点对之间新的最短路径将更长或代价更高。这可以作为评估单点脆弱性的一个考虑因素。

5.4.2　计算中间中心性得分

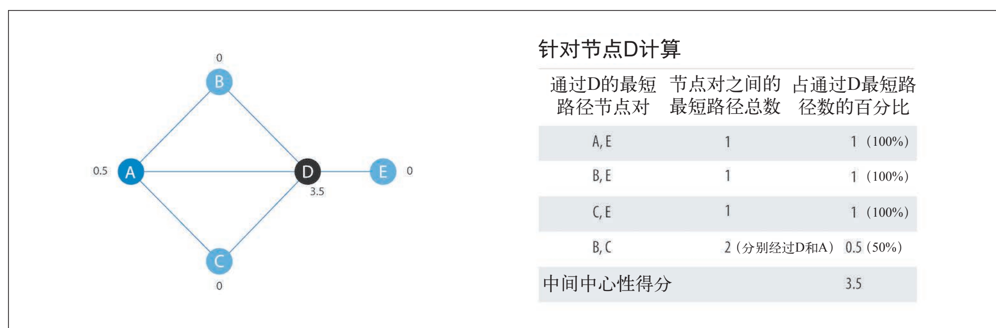
对所有最短路径，将下述公式的结果相加以计算节点的中间中心性得分：

$$B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p}$$

其中：

- u 为节点；
- p 为节点 s 和 t 之间最短路径的总数；
- $p(u)$ 为 s 与 t 之间通过节点 u 的最短路径的数量。

图 5-8 展示了计算中间中心性得分的步骤。



通过D的最短路径节点对	节点对之间的最短路径总数	占通过D最短路径数的百分比
A, E	1	1（100%）
B, E	1	1（100%）
C, E	1	1（100%）
B, C	2（分别经过D和A）	0.5（50%）
中间中心性得分		3.5

图 5-8：计算中间中心性得分的基本理念

计算过程如下。

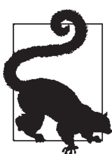
1. 对每个节点，找出通过它的最短路径。B、C、E 没有通过它们的最短路径，因此赋值为 0。
2. 对于步骤 1 中的每条最短路径，计算其占这对节点间可能最短路径数的百分比。
3. 将步骤 2 中的所有值相加，得到节点的中间中心性得分。图 5-8 中的列表展示了针对节点 D 执行步骤 2 和步骤 3 的过程。
4. 对每个节点重复该过程。

5.4.3　何时使用中间中心性算法

中间中心性算法适用于解决现实网络中的许多问题，比如查找瓶颈点、控制点和脆弱点。

示范用例如下。

- 识别各类组织中有影响力的人。有权势的人并不一定在管理岗位，使用中间中心性算法可能在“中间人岗位”上发现这样的人。撤去这些有影响力的人会严重削弱组织的活力。对于犯罪组织，抓获其头目是执法部门的目标，但如果企业失去了被低估的关键员工，对其而言将是一场灾难。Carlo Morselli 和 Julie Roy 的论文“Brokerage Qualifications in Ringing Operations”对此有更详细的介绍。
- 发现像电网这样的网络中的关键传输点。有点违背直觉的是，移除特定桥接点实际上可以隔离干扰，增强整体稳健性。Ricard Solé 等人在论文“Robustness of the European Power Grids Under Intentional Attack”中给出了更多研究细节。
- 通过针对有影响力者的推荐引擎，帮助用户在 Twitter 上传播影响力。Shanchan Wu 等人在论文“Making Recommendations in a Microblog to Improve the Impact of a Focal User”中描述了这种方法。



中间中心性算法假设节点之间的所有通信都是沿着最短路径进行的，并且频率相同，但在现实生活中并不总是这样。因此，它并没有为查找图中最有影响力的节点提供完美的方法，而是做了很好的示范。Mark Newman 在《网络科学引论》一书中有详细解释。

5.4.4　使用Neo4j实现中间中心性算法

Spark 没有内置中间中心性算法，所以在此使用 Neo4j 演示该算法。调用下面的程序将计算图中每个节点的中间中心性得分：

```
CALL algo.betweenness.stream("User", "FOLLOWS")
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id  AS user, centrality
ORDER BY centrality DESC
```

运行该程序，结果如下所示：

user	centrality
Alice	10.0
Doug	7.0
Mark	7.0
David	1.0
Bridget	0.0
Charles	0.0
Michael	0.0
Amy	0.0
James	0.0

如图 5-9 所示，Alice 是该网络中的主要中间人，而 Mark 和 Doug 也并不落后。在那个较小的子图中，所有最短路径都要经过 David，因此他对这些节点之间的信息流通非常重要。

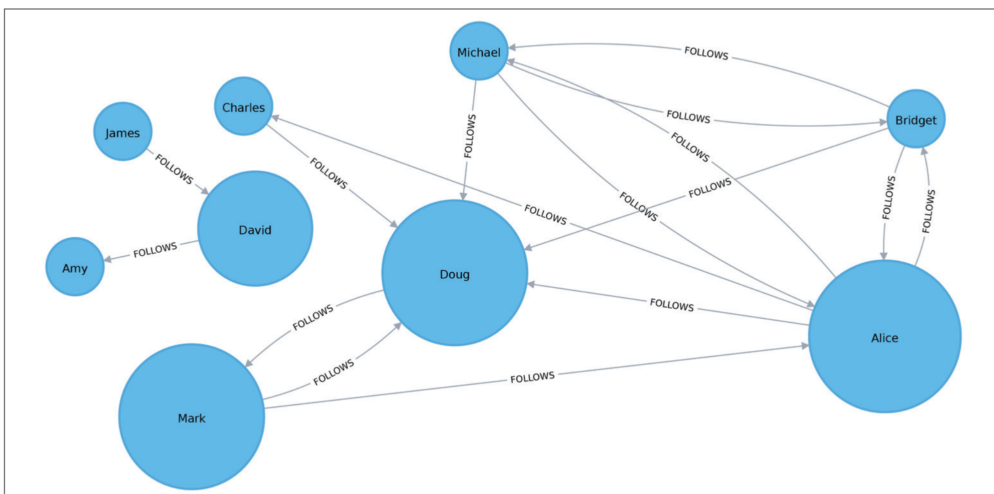


图 5-9：中间中心性的可视化



对于大型图，精确计算中心性得分并不实用。要精确计算所有节点的中间中心性得分，目前最快的算法的运行时间与节点数量和关系数量的乘积成正比。我们希望首先通过筛选形成子图，或者针对节点的子集应用这些算法（稍后介绍）。

可以将两个不连通的分量连接在一起，做法是引入一个新用户 Jason，来自这两组用户的人都会关注他或者被他关注：

```
WITH ["James", "Michael", "Alice", "Doug", "Amy"] AS existingUsers

MATCH (existing:User) WHERE existing.id IN existingUsers
MERGE (newUser:User {id: "Jason"})

MERGE (newUser)<-[:FOLLOWS]-(existing)
MERGE (newUser)-[:FOLLOWS]->(existing)
```

如果重新运行该算法，输出结果将如下所示：

user	centrality
Jason	44.33333333333333
Doug	18.333333333333332
Alice	16.666666666666664
Amy	8.0
James	8.0
Michael	4.0
Mark	2.1666666666666665
David	0.5
Bridget	0.0
Charles	0.0

Jason 的得分最高，这是因为两组用户通过他实现交流。可以把 Jason 视为这两组用户之间的**局部桥**，如图 5-10 所示。

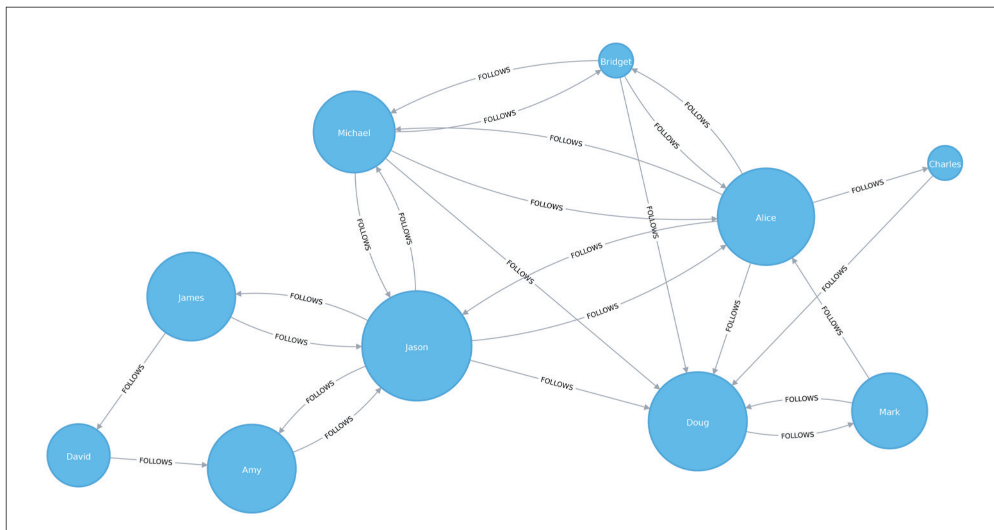


图 5-10：Jason 的中间中心性可视化

在继续介绍之前，先删除 Jason 及其关系以重置图。

```
MATCH (user:User {id: "Jason"})
DETACH DELETE user
```

5.4.5　中间中心性算法变体：RA-Brandes算法

回想一下，要在大型图上准确计算中间中心性得分，代价非常高。因此，可以选用一种运行速度快得多但仍然可以提供有用（尽管不精确）信息的近似算法。

RA-Brandes 算法是近似计算中间中心性得分的著名算法。它只考虑节点的一个子集，而不计算每对节点之间的最短路径。选择节点子集的两种常见策略如下。

- **随机方式**

  均匀、随机选取节点，选取概率确定。默认概率为 $\frac{\log_{10}(N)}{e^2}$。如果概率为 1，则算法与常规中间中心性算法一样，要加载全部节点。

- **度方式**

  随机选取节点，但自动排除那些度低于平均值的节点（只有具有大量关系的节点才有可能被访问）。

  要进一步优化，还可以限制最短路径算法的深度，使之提供全部最短路径的一个子集。

- 使用 Neo4j 实现中间中心性的近似算法

  下面的查询使用随机选取方式执行 RA-Brandes 算法：

```
CALL algo.betweenness.sampled.stream("User", "FOLLOWS", {strategy:"degree"})
YIELD nodeId, centrality
RETURN algo.getNodeById(nodeId).id AS user, centrality
ORDER BY centrality DESC
```

运行该程序，结果如下所示：

user	centrality
Alice	9.0
Mark	9.0
Doug	4.5
David	2.25
Bridget	0.0
Charles	0.0
Michael	0.0
Amy	0.0
James	0.0

虽然现在 Mark 排在 Doug 之前，但最具影响力的几个人物还是和以前差不多。

由于该算法具有随机性，因此每次的运行结果可能不同。相比小样本图，这种随机性对大型图的影响更大。

5.5 PageRank算法

PageRank 算法可能是最著名的中心性算法。它度量节点的传递性（或方向性）影响。前面讨论的其他所有中心性算法都是度量节点的直接影响，PageRank 算法则考虑节点的邻节点的影响，以及其邻节点的邻节点的影响。例如与拥有一大堆影响力小的朋友相比，拥有几个很有影响力的朋友能让人更有影响力。PageRank 算法的计算方法有两种：一种是将一个节点的等级迭代分散到其邻节点，另一种是随机遍历图并计算每个节点在遍历过程中被命中的频率。

PageRank 算法是以谷歌公司联合创始人 Larry Page 的姓氏命名的，Larry Page 创建 PageRank 算法的初衷是在谷歌的搜索结果中对网站进行评级。其基本假设是，一个网页如果有更多或更有影响力的输入链接，就更有可能是可信来源。PageRank 算法度量节点输入关系的数量和质量，以此估计该节点的重要性。在网络中，如果节点拥有的来自其他有影响力节点的输入关系越多，那么它就越可能在网络中占据主导地位。

5.5.1 影响力

直观地说，相对于那些不太重要的节点而言，与较为重要的节点相连对节点的影响更大。衡量影响力通常涉及对节点的评分，要用到加权关系，之后还要在多次迭代中更新分值。有时要对所有节点进行评分，但有时也可随机选取节点作为代表性样本。



请记住，中心性指标代表了某节点相对于其他节点的重要性。中心性是对节点潜在影响力的排序，而不是对实际影响力的度量。比如说，在网络中有两个人处于中心地位，但可能由于种种原因，其实际影响力会转移到其他人身上。量化实际影响力是设计附加影响指标中很活跃的一个研究领域。

5.5.2 PageRank算法公式

谷歌公司在最初的论文中将 PageRank 算法定义为：

$$PR(u) = (1-d) + d\left(\frac{PR(T1)}{C(T1)} + \cdots + \frac{PR(Tn)}{C(Tn)}\right)$$

解释如下。

- 假设页面 u 中含有从第 $T1$ 页到第 Tn 页的引用。
- d 是取值介于 0 ～ 1 的阻尼系数，通常设为 0.85。可以将其视为用户将持续单击的概率。这有助于最小化等级沉没，稍后解释。
- $1 - d$ 是不考虑任何关系直接到达节点的概率。
- $C(Tn)$ 定义为节点 T 的出度。

图 5-11 展示了 PageRank 算法如何持续更新节点等级，直到算法收敛或满足设置的迭代次数为止。

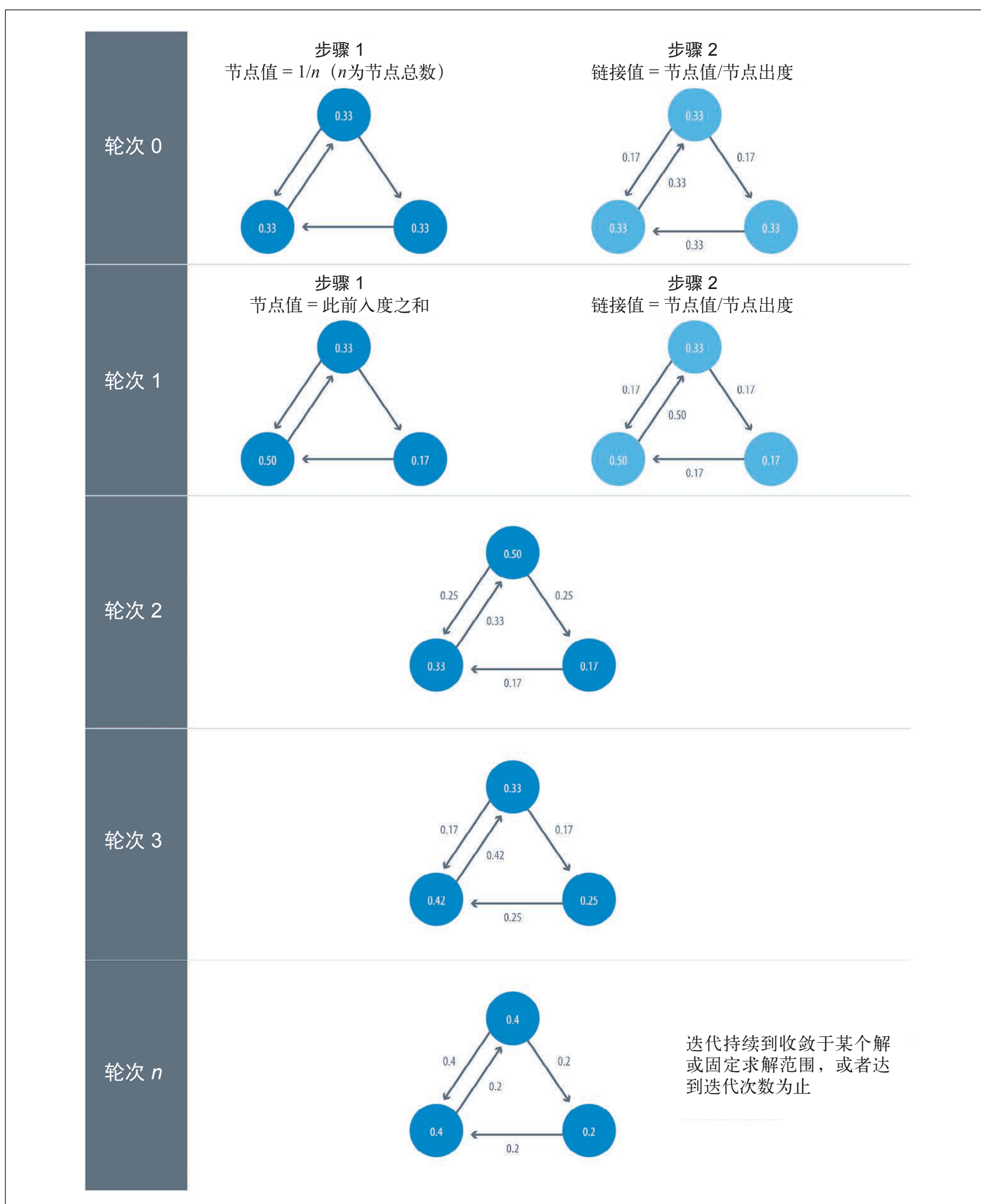


图 5-11：PageRank 算法的每次迭代都有两个计算步骤：一个更新节点值，另一个更新链接值

5.5.3 迭代、随机冲浪者和等级沉没

PageRank 算法是一种迭代算法，它要么运行到分值收敛为止，要么运行到一定迭代次数为止。

从概念上讲，PageRank 算法假定有一个 Web 冲浪者通过链接或随机 URL 访问页面。阻尼系数

d 定义了下一次通过该链接单击的概率。可以将其视为 Web 冲浪者感到无聊并随机切换到另一个页面的概率。PageRank 评分反映了通过输入链接访问而非随机访问一个页面的可能性。

没有输出关系的节点（也称**悬挂节点**）或节点分组可以通过拒绝共享而独占 PageRank 得分，这就是所谓的**等级沉没**。可以把这想象成一个 Web 冲浪者被困在某个页面或者页面子集中没有出路。另一个难题是由分组中的各节点仅相互指向对方而造成的。当 Web 冲浪者在节点之间来回跳转时，循环引用会导致其等级升高。这些情况如图 5-12 所示。

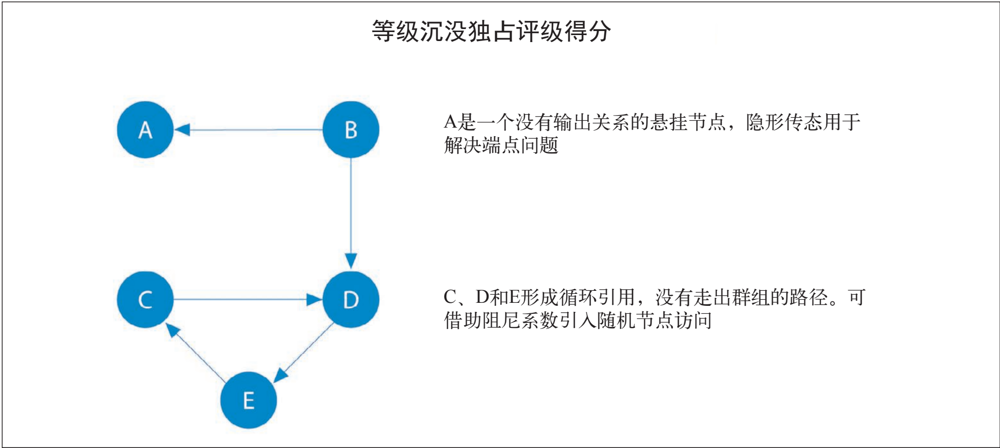


图 5-12：等级沉没是因为一个节点或节点分组没有输出关系

有两种策略可以避免等级沉没。首先，当到达一个没有输出关系的节点时，PageRank 算法假设它到所有节点都有输出关系。穿越这些看不见的关联关系有时也被称为**隐形传态**。其次，阻尼系数提供了另一种避免等级沉没的方法，相对于直接链接，引入了随机节点访问的概率。当把 *d* 设为 0.85 时，完全访问随机节点的概率为 15%。

虽然原始公式建议阻尼系数取 0.85，但这起初适用于万维网上链接的幂律分布（大多数页面的链接很少，少数页面有很多链接）。降低阻尼系数会降低在随机跳转之前沿着长关系路径前进的可能性，反之将增加节点的前一个节点对其得分和评级的贡献。

如果通过 PageRank 算法得到了意外的结果，那么有必要对该图进行探索性分析，以判断原因是否源自上述这些问题。可以阅读 Ian Rogers 的文章“The Google PageRank Algorithm and How It Works”了解更多内容。

5.5.4 何时使用PageRank算法

PageRank 算法现已用于 Web 索引之外的诸多领域。当在网络上寻找具有广泛影响力的节点时，可使用该算法，例如当你想寻找对生物功能整体影响最大的基因时，可能找到的并非关联关系最多的基因，而是与其他重要功能联系最密切的基因。

示范用例如下。

- 向用户推荐他们可能感兴趣的账户（Twitter 为此使用了个性化 PageRank 算法）。该算法可在含有共享兴趣爱好和公共连接的图上运行。
- 预测公共空间或街道的交通流量和人类活动。该算法可在含有道路交叉口的图上运行，其中 PageRank 评分反映了人们在每条街道上停车或结束旅程的意向。Bin Jiang、Sijian Zhao 和 Junjun Yin 的论文“Self-Organized Natural Roads for Predicting Traffic Flow: A Sensitivity Study”对此有详细阐述。
- 应用于医疗行业和保险行业的异常和欺诈检测系统。PageRank 算法有助于揭示医生或供应商的异常行为，然后可以将得到的分值输入机器学习算法中。

David Gleich 在论文“PageRank Beyond the Web”中介绍了该算法的更多应用。

5.5.5 使用Spark实现PageRank算法

下面准备运行 PageRank 算法。GraphFrames 支持两种 PageRank 算法实现。

- 第 1 个实现以固定次数迭代运行 PageRank 算法。这可以通过设置 `maxIter` 参数来完成。
- 第 2 个实现运行 PageRank 算法，直至收敛。这可以通过设置 `tol` 参数来完成。

1. 具有固定迭代次数的 PageRank 算法

看一个采用固定迭代方法的例子：

```
results = g.pageRank(resetProbability=0.15, maxIter=20)
results.vertices.sort("pagerank", ascending=False).show()
```

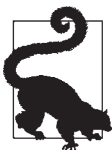


注意，在 Spark 中将阻尼系数直观地称作**重置概率**，其值为反向值。换言之，本例中的 `resetProbability=0.15` 等价于 Neo4j 中的 `dampingFactor:0.85`。

在 PySpark 中运行以上代码，输出结果如下所示：

id	pageRank
Doug	2.2865372087512252
Mark	2.1424484186137263
Alice	1.520330830262095
Michael	0.7274429252585624
Bridget	0.7274429252585624
Charles	0.5213852310709753
Amy	0.5097143486157744
David	0.36655842368870073
James	0.1981396884803788

正如预期，Doug 的 PageRank 得分最高，这是因为其子图中所有用户都关注他。虽然 Mark 只有一个关注者，但因为关注者是 Doug，所以 Mark 在该图中也很重要。重要的不仅是关注者的数量，还有关注者自身的重要程度。



运行 PageRank 算法的图并没有对关系加权，因此每个关系都是平等的。通过在关系 DataFrame 中指定 `weight` 列，可以为关系添加权重。

2. 运行 PageRank 算法，直至收敛

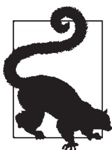
下面试试收敛方式的算法实现，它将运行 PageRank 算法，直到结束于容差范围内的某个解：

```
results = g.pageRank(resetProbability=0.15, tol=0.01)
results.vertices.sort("pagerank", ascending=False).show()
```

在 PySpark 中运行这段代码，输出结果如下所示：

id	pageRank
Doug	2.2233188859989745
Mark	2.090451188336932
Alice	1.5056291439101062
Michael	0.733738785109624
Bridget	0.733738785109624
Amy	0.559446807245026
Charles	0.5338811076334145
David	0.40232326274180685
James	0.21747203391449021

每个人的 PageRank 得分与固定迭代次数的算法实现略有不同，但是正如预期，排序仍然基本相同。



尽管最理想的情况是收敛于最优解，但是在某些情况下，PageRank 算法无法做到数学上的收敛。对于大型图，PageRank 算法的运行时间可能会非常长。采用容差限制有助于为收敛结果设置一个可接受的范围，但是许多人选择用（或结合这种方法）最大迭代次数方式来代替。最大迭代设置通常可以提供更好的性能一致性。无论选择哪种方式，都需要测试不同边界才能找到适合数据集的方法。与中型图相比，大型图通常需要更多迭代次数或更小的容差范围才能获得更高的精度。

5.5.6　使用Neo4j实现PageRank算法

还可以在 Neo4j 平台上运行 PageRank 算法。调用下面的程序将计算图中每个节点的 PageRank 得分：

```
CALL algo.pageRank.stream('User', 'FOLLOWS', {iterations:20, dampingFactor:0.85})
YIELD nodeId, score
RETURN algo.getNodeById(nodeId).id AS page, score
ORDER BY score DESC
```

运行该程序，结果如下所示：

page	score
Doug	1.6704119999999998
Mark	1.5610085
Alice	1.1106700000000003
Bridget	0.535373
Michael	0.535373
Amy	0.385875
Charles	0.3844895
David	0.2775
James	0.15000000000000002

与 Spark 示例一样，Doug 是其中最有影响力的用户，Mark 紧随其后，他是 Doug 唯一关注的用户。图 5-13 展现了节点的相对重要性。

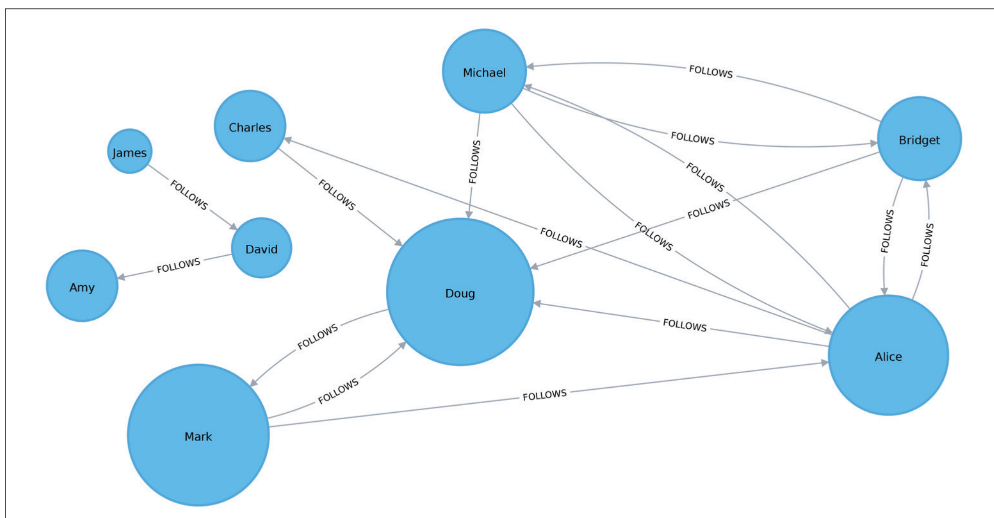


图 5-13：PageRank 算法的可视化



PageRank 算法的实现各不相同，因此即使排序相同，评分也可能不同。Neo4j 用 1 减去阻尼系数得到的值来初始化节点，而 Spark 使用 1 来初始化。在这种情况下，得到的相对等级（PageRank 算法的目标）是相同的，但用于得出这些结果的基础评分值不同。



与 Spark 示例一样，这里在运行 PageRank 算法时，图中的关系没有加权，因此每个关系都是平等的。在传递给 PageRank 程序的配置中包含 `weightProperty` 属性，通过该属性可以设置关系权重。例如关系的属性 `weight` 代表权重，可以把下述配置传递给程序：`weightProperty: "weight"`。

5.5.7 PageRank算法变体：个性化PageRank算法

个性化 PageRank（personalized PageRank，PPR）是 PageRank 算法的一种变体，它从特定节点出发，计算图中节点的重要程度。对于 PPR，随机跳转指的是返回到给定的起始节点集。这种偏向性结果（或者说个性化）针对的是起始节点。这种偏向和本地化使得 PPR 对于靶向性强的推荐非常有用。

使用 Spark 实现个性化 PageRank 算法

可以通过传递 `sourceId` 参数来计算给定节点的 PPR 得分。下面的代码计算 Doug 的 PPR 得分：

```
me = "Doug"
results = g.pageRank(resetProbability=0.15, maxIter=20, sourceId=me)
people_to_follow = results.vertices.sort("pagerank", ascending=False)

already_follows = list(g.edges.filter(f"src = '{me}'").toPandas()["dst"])
people_to_exclude = already_follows + [me]

people_to_follow[~people_to_follow.id.isin(people_to_exclude)].show()
```

该查询结果可用于向 Doug 推荐可关注的人。请注意，还要确保在最终结果中排除 Doug 已经关注的人和他自己。

在 PySpark 中运行以上代码，输出结果如下所示：

id	pageRank
Alice	0.1650183746272782
Michael	0.048842467744891996
Bridget	0.048842467744891996
Charles	0.03497796119878669
David	0.0
James	0.0
Amy	0.0

Alice 是 Doug 应该关注的最佳人选，当然，也可以推荐 Michael 和 Bridget。

5.6 小结

对于在网络中识别有影响力的节点，中心性算法是强大的工具。本章介绍了典型的中心性算法：度中心性算法、接近中心性算法、中间中心性算法和 PageRank 算法，还讨论了用于处理长时间运行和隔离分量等问题的几种算法变体以及替代选项。

中心性算法用途广泛，推荐将其应用于各类分析。你可以应用所学知识来定位传播信息的最佳接触点，找到控制资源流动的隐藏中间点，并且发现躲在幕后的势力。

接下来介绍用于研究群组和分割的社团发现算法。

第 6 章

社团发现算法

社团的形成在所有类型的网络中都很常见，识别社团对于评价群体行为和突发现象不可或缺。发现社团的一般性原则是，社团成员在群组内部的关系要多于其与群组外部节点的关系。识别这些有关联关系的集合揭示了节点簇、孤立群组和网络结构。这些信息有助于推断同类群组的相似行为或偏好，评估弹性，查找嵌套关系，并为其他分析准备数据。社团发现算法也常用于实现面向常规检测的网络可视化。

本章将详细介绍几种最具代表性的社团发现算法。

- 面向整体关系稠密度的三角形计数和聚类系数。
- 用于发现连通簇的强连通分量算法和连通分量算法。
- 标签传播算法，可基于节点标签快速推断群组。
- Louvain 模块度算法，用于研究分组的质量和层级结构。

本章将解释这些算法的工作原理，并给出相应的 Spark 示例和 Neo4j 示例。当算法仅适用于一种平台时，仅提供一个示例。本章还会用到加权关系，这是因为权重通常用于表示不同关系的重要程度。

图 6-1 展示了各种社团发现算法之间的差异，表 6-1 是每种算法及其示范用例的速查表。

度量算法

三角形计数
经过某一节点的三角形数量。经过节点A的三角形有两个

聚类系数
某节点的邻节点相互连接的可能性
A的聚类系数为0.2，也就是说，任意两个连接到A的节点都有20%的概率相互连接
这些指标可以全局计数或归一化

分量算法

连通分量算法
不考虑方向，集合中的所有节点都能到达集合中其他节点
图中由虚线圈出的两个集合：{A, B, C, D, E}和{F, G}

强连通分量算法
集合中的所有节点都可以双向到达其他各节点，但不必直接到达，如图中阴影部分

标签传播算法

为查找簇而将标签扩散给邻节点或接收来自邻节点的标签

要执行多次迭代
通常用关系或节点的权重来判断标签在群组中的“声望”

Louvain模块度算法

将节点移入更高一层关系稠密的群组，从而合并形成超级社团，以此查找簇

要进行多次迭代
使用关系权重及其总和来确定分组方式

图 6-1：有代表性的社团发现算法

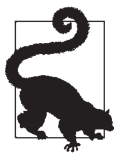


前文交替使用了**集合**（set）、**分割**（partition）、**簇**（cluster）、**群组**（group）和**社团**（community）等术语。这些术语表明，可以用不同的方法对相似节点进行分组。社团发现算法也称**聚类算法**和**分割算法**。本书针对特定算法一般使用文献中最常用的术语。

表6-1：社团发现算法总览

算法类型	作　　用	示范用例	Spark 示例	Neo4j 示例
三角形计数和聚类系数	度量有多少节点形成了三角形，以及节点的聚类程度	估计群组稳定性以及网络是否可能呈现“小世界”特征，在图中出现紧密关联的簇	有	有
强连通分量算法	查找每个节点都可以按照关系方向到达同群组其他各节点的群组	基于群组的从属关系或相似项进行产品推荐	有	有
连通分量算法	查找每个节点都可以到达同群组其他各节点的群组，而不考虑关系的方向	为其他算法实现快速分组，识别孤岛	有	有
标签传播算法	基于相邻多数节点扩散标签，借此推断出簇	理解社团的共识或者发现一起开的药物中可能存在的危险组合	有	有
Louvain 模块度算法	通过将关系权重和稠密度与某个确定的估计值或平均值做比较，最大化分组操作的预设精度	在诈骗分析中，评估一个群体是只有零星的几次不良行为还是有团伙诈骗行为	无	有

首先介绍示例数据并演示如何将数据导入 Spark 和 Neo4j，然后按表 6-1 中的顺序依次介绍这些算法：概述之后给出关于其使用场景的建议，还有何时使用相关算法的指南。理论先行，而后用样本数据演示示例代码。



在使用社团发现算法时，要注意关系的稠密度。

如果图非常稠密，那么很可能会导致所有节点聚集在一个或几个簇中。可以通过度、关系权重或相似度等指标来消除这种影响。

相反，如果图过于稀疏，连接的节点太少，则可能会导致每个节点各自成簇。在这种情况下，应尝试合并带有更多相关性信息的附加关系类型。

6.1　示例数据：软件依赖图

依赖图特别适合揭示社团发现算法之间的细微差异，这是因为它们往往更具关联性和层次性。依赖图可用于软件、电网等多种领域，本章示例运行所针对的图包含 Python 库之间的依赖关系，见表 6-2 和表 6-3。开发人员使用这种软件依赖图来追踪软件项目中的传递依赖和冲突。本书配套文件包含相应节点和文件。

表6-2：sw-nodes.csv

id
six
pandas
numpy

（续）

id
python-dateutil
pytz
pyspark
matplotlib
spacy
py4j
jupyter
jpy-console
nbconvert
ipykernel
jpy-client
jpy-core

表6-3：sw-relationships.csv

src	dst	relationship
pandas	numpy	DEPENDS_ON
pandas	pytz	DEPENDS_ON
pandas	python-dateutil	DEPENDS_ON
python-dateutil	six	DEPENDS_ON
pyspark	py4j	DEPENDS_ON
matplotlib	numpy	DEPENDS_ON
matplotlib	python-dateutil	DEPENDS_ON
matplotlib	six	DEPENDS_ON
matplotlib	pytz	DEPENDS_ON
spacy	six	DEPENDS_ON
spacy	numpy	DEPENDS_ON
jupyter	nbconvert	DEPENDS_ON
jupyter	ipykernel	DEPENDS_ON
jupyter	jpy-console	DEPENDS_ON
jpy-console	jpy-client	DEPENDS_ON
jpy-console	ipykernel	DEPENDS_ON
jpy-client	jpy-core	DEPENDS_ON
nbconvert	jpy-core	DEPENDS_ON

图 6-2 展示了我们要构造的图，该图中有 3 个关于库的簇。可以将小规模数据集可视化，作为辅助工具验证由社团发现算法推导出来的簇。

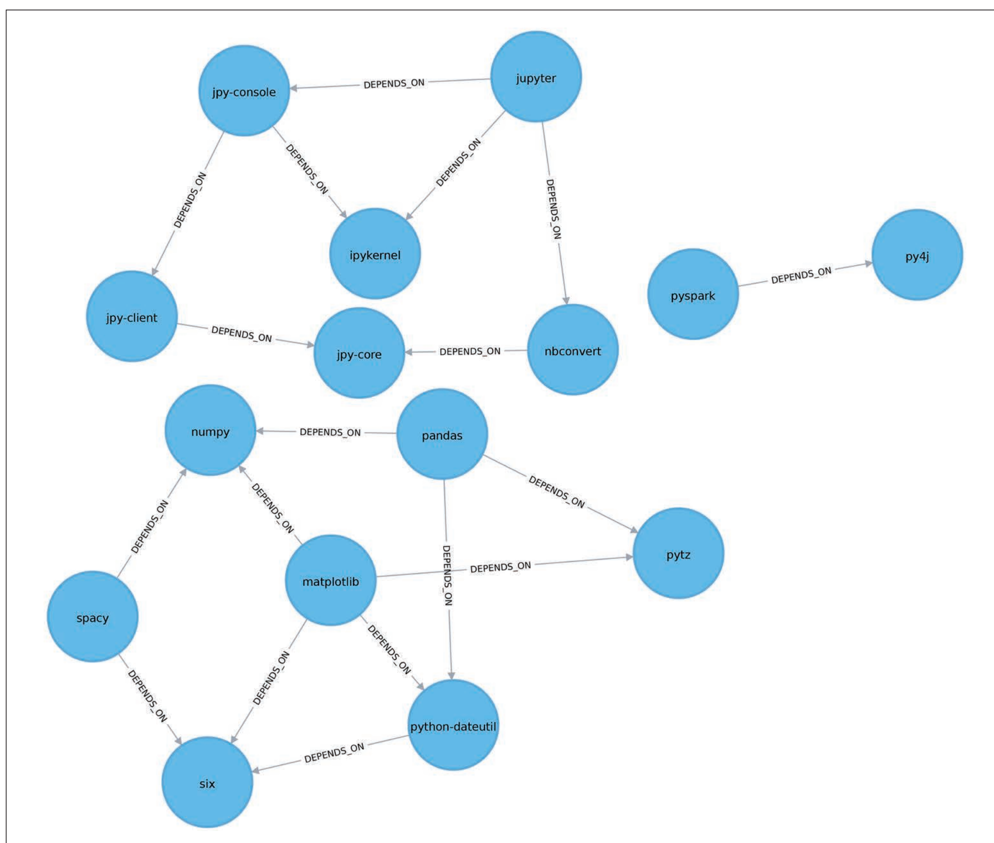


图 6-2：示例图模型

下面基于示例 CSV 文件在 Spark 和 Neo4j 中创建图。

6.1.1　将数据导入Spark

首先从 Spark 和 GraphFrames 包中导入所需的包：

```
from graphframes import *
```

下面的函数从示例 CSV 文件中创建一个 GraphFrame 对象：

```
def create_software_graph():
    nodes = spark.read.csv("data/sw-nodes.csv", header=True)
    relationships = spark.read.csv("data/sw-relationships.csv", header=True)
    return GraphFrame(nodes, relationships)
```

然后调用该函数。

```
g = create_software_graph()
```

6.1.2 将数据导入Neo4j

接下来对 Neo4j 执行相同的操作。下面的查询导入各节点：

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "sw-nodes.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MERGE (:Library {id: row.id})
```

下面的代码导入关系。

```
WITH "https://github.com/neo4j-graph-analytics/book/raw/master/data/" AS base
WITH base + "sw-relationships.csv" AS uri
LOAD CSV WITH HEADERS FROM uri AS row
MATCH (source:Library {id: row.src})
MATCH (destination:Library {id: row.dst})
MERGE (source)-[:DEPENDS_ON]->(destination)
```

现在图已加载完毕，下面开始算法层面的工作。

6.2 三角形计数和聚类系数

由于三角形计数和聚类系数经常同时使用，因此一并介绍。三角形计数算法确定了图中经过每个节点的三角形数量。三角形是由三个节点组成的集合，且每个节点与其他各节点都有关系。三角形计数算法也可以全局运行，用于评估整个数据集。



三角形数量多的网络更有可能呈现出小世界结构。

聚类系数的目标是比较群组的聚合紧密程度与其能够达到的聚合紧密程度。该算法在计算过程中用到了三角形计数算法，它提供现有三角形数与可能关系数的比值。最大值 1 表示团的每个节点都可以连接到该团的其他节点。

聚类系数有两类：局部聚类系数和全局聚类系数。

6.2.1 局部聚类系数

一个节点的局部聚类系数体现的是其邻节点也相互连通的可能性。该数值的计算过程要用到三角形计数算法。

节点聚类系数的计算方法如下：将通过该节点的三角形数乘 2，该群组的最大关系数减 1 后与最大关系数相乘，然后用前一个积除以后一个积。图 6-3 给出的各例中都有一个节点有 5 个关系，但是三角形数和聚类系数不同。

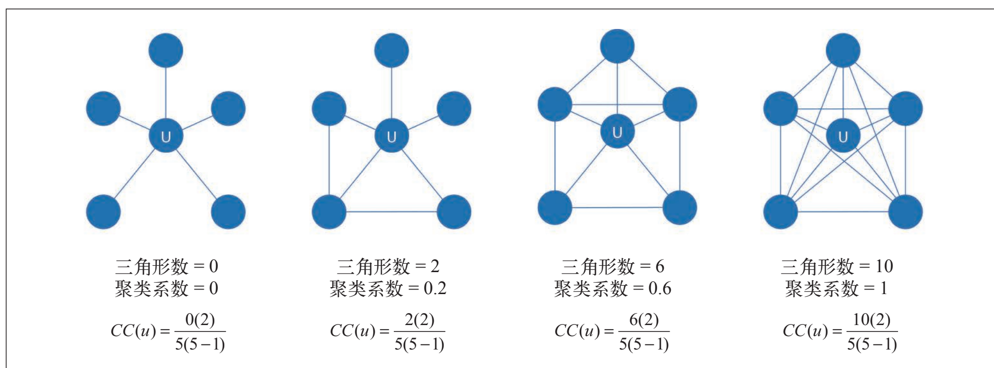


图 6-3：节点 U 的三角形计数和聚类系数

注意，图 6-3 中的节点 U 具有 5 个关系，这使得聚类系数总是等于三角形数量的 10%。当改变关系数量时，情况就不同了。如果将图 6-3 中的第 2 个例子改为节点 U 只有 4 个关系（同样有两个三角形），那么系数为 0.33。

求取节点的聚类系数可采用如下公式：

$$CC(u) = \frac{2R_u}{k_u(k_u - 1)}$$

其中：

- u 为节点；
- R_u 是通过 u 邻节点的关系数（可借助经过 u 的三角形数得到）；
- k_u 为 u 的度。

6.2.2 全局聚类系数

全局聚类系数是局部聚类系数的归一化和。

聚类系数为查找像团这样明显的群组提供了一种有效方法。在团中，每个节点都与其他各节点有关系，不过也可以指定阈值来设置级别（例如节点 40% 连通）。

6.2.3 何时使用三角形计数和聚类系数

当需要判断群组的稳定性时，可使用三角形计数算法，也可在计算其他网络指标（如聚类系数）时使用。三角形计数算法在社交网络分析中很流行，主要用于发现社团。

聚类系数可以反映随机选定节点的连通概率。还可以用聚类系数来快速评估特定群组或整个网络的内聚力。将这两种算法结合使用可估计弹性和探查网络结构。

示范用例如下。

- 在识别垃圾网站内容时找出特征。Luca Becchetti 等人的论文“Efficient Semi-Streaming Algorithms for Local Triangle Counting in Massive Graphs”对此有介绍。
- 研究 Facebook 社交图的社团结构，比如一些研究人员在一张看起来稀疏的全球社交图中发现了稠密的用户友邻关系。
- 探测 Web 的主题结构，并且基于网页间的相互链接来发现网页社团的共同主题。更多相关信息，参见 Jean-Pierre Eckmann 和 Elisha Moses 的文章“Curvature of Co-Links Uncovers Hidden Thematic Layers in the World Wide Web”。

6.2.4 使用Spark实现三角形计数算法

准备运行三角形计数算法，实现代码如下：

```
result = g.triangleCount()
(result.sort("count", ascending=False)
 .filter('count > 0')
 .show())
```

在 PySpark 中运行这段代码，输出结果如下所示：

count	id
1	jupyter
1	python-dateutil
1	six
1	ipykernel
1	matplotlib
1	jpy-console

结果表明一个节点的两个邻节点也相邻。6 个库都位于这样的三角形中。

如果还想知道这些三角形中都有哪些节点，就要涉及三角形流了，这需要使用 Neo4j 来实现。

6.2.5 使用Neo4j实现三角形计数算法

使用 Spark 无法获取三角形流，而使用 Neo4j 可以返回三角形流：

```
CALL algo.triangle.stream("Library","DEPENDS_ON")
YIELD nodeA, nodeB, nodeC
RETURN algo.getNodeById(nodeA).id AS nodeA,
       algo.getNodeById(nodeB).id AS nodeB,
       algo.getNodeById(nodeC).id AS nodeC
```

运行这段程序，结果如下所示：

nodeA	nodeB	nodeC
matplotlib	six	python-dateutil
jupyter	jpy-console	ipykernel

可以看到和之前一样的 6 个库，但现在知道它们是如何连接的了。matplotlib、six 和 python-dateutil 构成一个三角形，jupyter、jpy-console 和 ipykernel 构成了另一个三角形。图 6-4 展示了这些三角形。

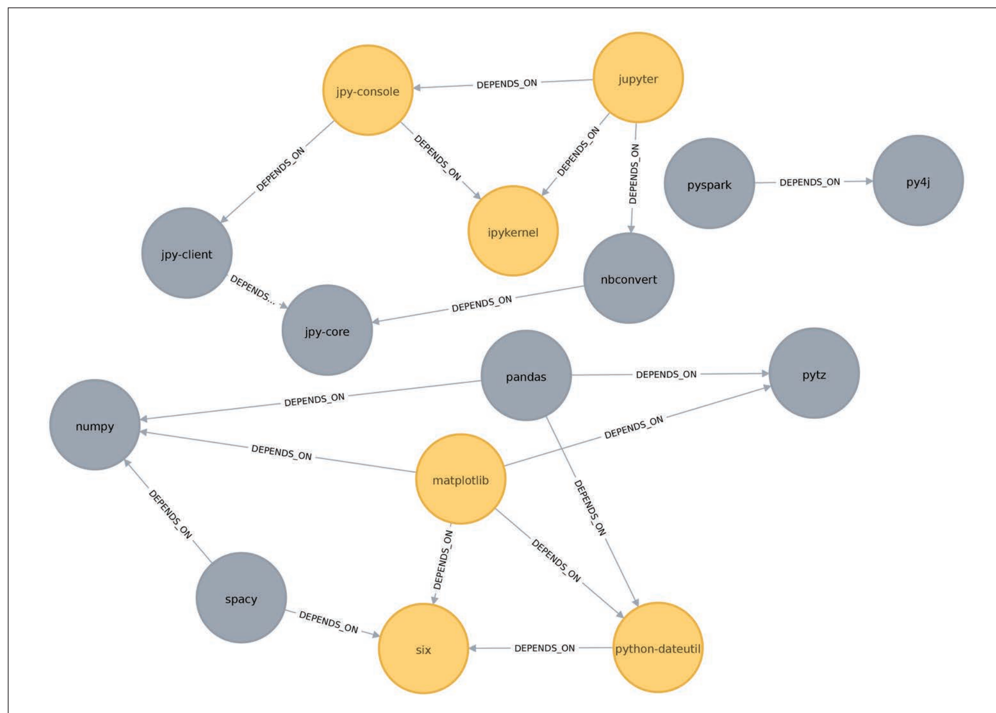


图 6-4：软件依赖图中的三角形

6.2.6 使用Neo4j计算局部聚类系数

还可以计算局部聚类系数。以下查询将针对每个节点计算局部聚类系数：

```
CALL algo.triangleCount.stream('Library', 'DEPENDS_ON')
YIELD nodeId, triangles, coefficient
WHERE coefficient > 0
RETURN algo.getNodeById(nodeId).id AS library, coefficient
ORDER BY coefficient DESC
```

运行该程序，结果如下所示：

library	coefficient
ipykernel	1.0
jupyter	0.3333333333333333
jpy-console	0.3333333333333333
six	0.3333333333333333
python-dateutil	0.3333333333333333
matplotlib	0.16666666666666666

ipykernel 的得分为 `1.0`，这意味着它的所有邻节点都彼此相邻。图 6-4 清楚地展现了这一点。这表明，直接围绕 ipykernel 形成的社团非常有内聚力。

该代码示例过滤了系数为 0 的节点，但是系数较小的节点可能也会有用。得分低可能标志着节点为**结构洞**，即与多个社团中的节点（相互之间没有连接）都有良好连接的节点。这也是第 5 章讨论的寻找**潜在**桥的方法。

6.3　强连通分量算法

强连通分量算法（strongly connected components algorithm，SCC algorithm）是最早的图算法之一。SCC 在一个有向图中查找连通的节点集，其中每个节点都与同一集合中其他任何节点双向可达。该算法运行时，操作规模与节点数成正比。图 6-5 显示，SCC 群组中的节点并不需要直接相邻，但是在集合中的所有节点之间必须存在有向路径。

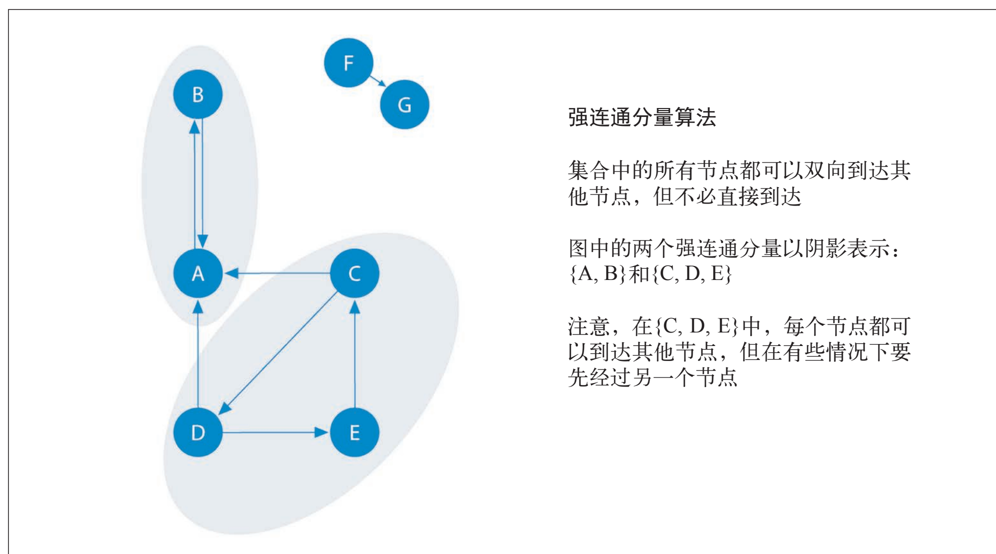


图 6-5：强连通分量算法



将有向图分解为它的强连通分量是深度优先搜索算法的经典应用之一。Neo4j 在实现强连通分量算法时在底层要用到深度优先搜索算法。

6.3.1　何时使用强连通分量算法

强连通分量算法用于在图分析的早期步骤中了解图的构造，或者用于识别可能需要单独研究的紧密关联簇。对于像推荐引擎这样的应用，强连通分量算法可用于分析群组的相似行为或趋势。

许多像强连通分量算法这样的社团发现算法用于发现簇，或者将簇分解成单个节点以进行簇间的分析。也可以使用强连通分量算法来可视化环，以进行如查找死锁进程这样的分析，出现死锁是因为每个子进程都在等待其他子进程的动作。

示范用例如下。

- 找出每个成员直接或间接拥有其他成员股份的公司集合。
- 在多跳无线网络中测量路由性能时，计算不同网络配置的连通性。可以阅读 Mahesh K. Marina 和 Samir R. Das 的论文“Routing Performance in the Presence of Unidirectional Links in Multihop Wireless Networks”，了解更多内容。
- 在许多仅对强连通图有效的图算法中充当第一步。在社交网络中，我们发现了许多强连通的群组，这些群组中的人通常都有相似的偏好，使用强连通分量算法可以查找此类群组，并向群组中的人推荐其可能喜欢的网页或想购买的产品。



尽管有些算法采用了可避免无限循环的策略，但是在自己编写算法或寻找无法终止的进程时，也可以使用强连通分量算法来检查环。

6.3.2　使用Spark实现强连通分量算法

启动 Spark，首先从 Spark 和 GraphFrames 包中导入所需的包：

```
from graphframes import *
from pyspark.sql import functions as F
```

然后准备运行强连通分量算法，用它来判断图中是否存在循环依赖关系。



如果两个节点之间存在双向路径，那么它们只能出现在同一强连通分量中。

编写如下代码来运行算法：

```
result = g.stronglyConnectedComponents(maxIter=10)
(result.sort("component")
 .groupby("component")
 .agg(F.collect_list("id").alias("libraries"))
 .show(truncate=False))
```

在 PySpark 中运行这段代码，输出结果如下所示：

component	libraries
180388626432	[jpy-core]
223338299392	[spacy]
498216206336	[numpy]
523986010112	[six]
549755813888	[pandas]
558345748480	[nbconvert]
661424963584	[ipykernel]
721554505728	[jupyter]
764504178688	[jpy-client]
833223655424	[pytz]
910533066752	[python-dateutil]
936302870528	[pyspark]
944892805120	[matplotlib]
1099511627776	[jpy-console]
1279900254208	[py4j]

每个库（节点）都被分配到唯一的分量中了。这是它所属的分割或子群组，正如我们预料的那样，每个节点都在自己的分割中。这意味着该软件项目的这些库之间不存在循环依赖关系。

6.3.3 使用Neo4j实现强连通分量算法

使用 Neo4j 运行强连通分量算法，执行以下查询即可：

```
CALL algo.scc.stream("Library", "DEPENDS_ON")
YIELD nodeId, partition
RETURN partition, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

算法传递的参数如下。

- ❑ `Library`

  从图加载的节点标签。

- ❑ `DEPENDS_ON`

  从图加载的关系类型。

该查询的返回结果如下所示：

partition	libraries
8	[ipykernel]
11	[six]
2	[matplotlib]
5	[jupyter]
14	[python-dateutil]
13	[numpy]
4	[py4j]
7	[nbconvert]
1	[pyspark]
10	[jpy-core]
9	[jpy-client]
3	[spacy]
12	[pandas]
6	[jpy-console]
0	[pytz]

与 Spark 示例一样，每个节点都在自己的分割中。

到目前为止，该算法表明各个 Python 库都表现良好。不过，可以在图中创建一个循环依赖项，让研究过程变得更有趣。这意味着最终在同一个分割中会有多个节点。

下面的查询添加了一个名为 extra 的库，它在 py4j 和 pyspark 之间创建了循环依赖关系：

```
MATCH (py4j:Library {id: "py4j"})
MATCH (pyspark:Library {id: "pyspark"})
MERGE (extra:Library {id: "extra"})
MERGE (py4j)-[:DEPENDS_ON]->(extra)
MERGE (extra)-[:DEPENDS_ON]->(pyspark)
```

图 6-6 清晰地展示了创建的循环依赖关系。

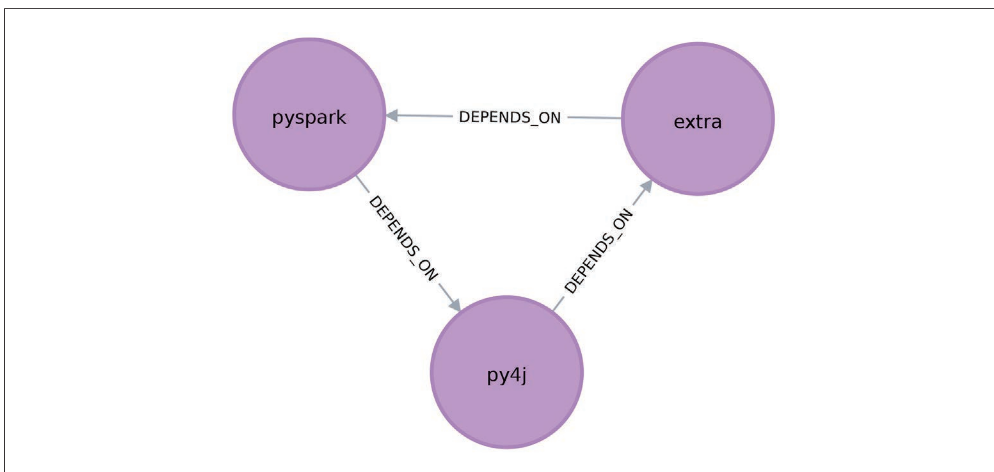


图 6-6：pyspark、py4j 和 extra 之间的循环依赖关系

再次运行强连通分量算法，结果稍有不同，如下所示：

partition	libraries
1	[pyspark, py4j, extra]
8	[ipykernel]
11	[six]
2	[matplotlib]
5	[jupyter]
14	[numpy]
13	[pandas]
7	[nbconvert]
10	[jpy-core]
9	[jpy-client]
3	[spacy]
15	[python-dateutil]
6	[jpy-console]
0	[pytz]

pyspark、py4j 和 extra 都属于同一分割，强连通分量算法帮我们找到了循环依赖项！

在介绍下一个算法之前，先删除图中的 extra 库及其关系。

```
MATCH (extra:Library {id: "extra"})
DETACH DELETE extra
```

6.4　连通分量算法

连通分量算法（connected components algorithm，也称**联盟查找算法**或**弱连通分量算法**）可在无向图中发现连通节点集合。与强连通分量算法不同，该算法只需要节点对之间存在单向路径，而强连通分量算法需要节点对之间存在双向路径。Bernard A. Galler 和 Michael J. Fisher 在 1964 年发表的论文“An Improved Equivalence Algorithm”中首次阐述了这种算法。

6.4.1　何时使用连通分量算法

与强连通分量算法一样，连通分量算法通常在分析的早期使用，用于理解图的结构。因为该算法的伸缩性强，所以可用于需要频繁更新的图。该算法可以快速显示群组间共同的新节点，这对如欺诈检测这样的分析来说非常有用。

应该养成习惯，将运行连通分量算法作为常规图分析的准备步骤，测试图是否连通。执行这一快速测试可以避免意外地在图的某个非连通分量上运行算法，导致结果出错。

示范用例如下。

- 追踪数据库记录簇，作为数据去重过程的一部分。去重是主数据管理应用中的一项重要任务，详见 Alvaro Monge 和 Charles Elkan 的论文“An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records”。
- 分析引文网络。这项研究使用连通分量算法来了解网络连通情况，然后看看如果从图中移走“中心”节点或“权威”节点，能否保持连通性。Yuan An、Jeannette Janssen 和 Evangelos E. Milios 在论文“Characterizing and Mining the Citation Graph of the Computer Science Literature”中进一步阐述了该用例。

6.4.2　使用Spark实现连通分量算法

首先启动 Spark，首先从 Spark 和 GraphFrames 包中导入所需的包：

```
from pyspark.sql import functions as F
```



如果两个节点之间存在双向路径，则它们位于同一连通分量中。

然后运行连通分量算法，代码如下：

```
result = g.connectedComponents()
(result.sort("component")
 .groupby("component")
 .agg(F.collect_list("id").alias("libraries"))
 .show(truncate=False))
```

在 PySpark 中运行以上代码，输出结果如下所示：

component	libraries
180388626432	[jpy-core, nbconvert, ipykernel, jupyter, jpy-client, jpy-console]
223338299392	[spacy, numpy, six, pandas, pytz, python-dateutil, matplotlib]
936302870528	[pyspark, py4j]

结果显示有 3 个节点簇，如图 6-7 所示。本例中仅有 3 个分量，一目了然。对于大型图来说，这种算法更有价值，因为要么肉眼难以识别，要么非常耗时。

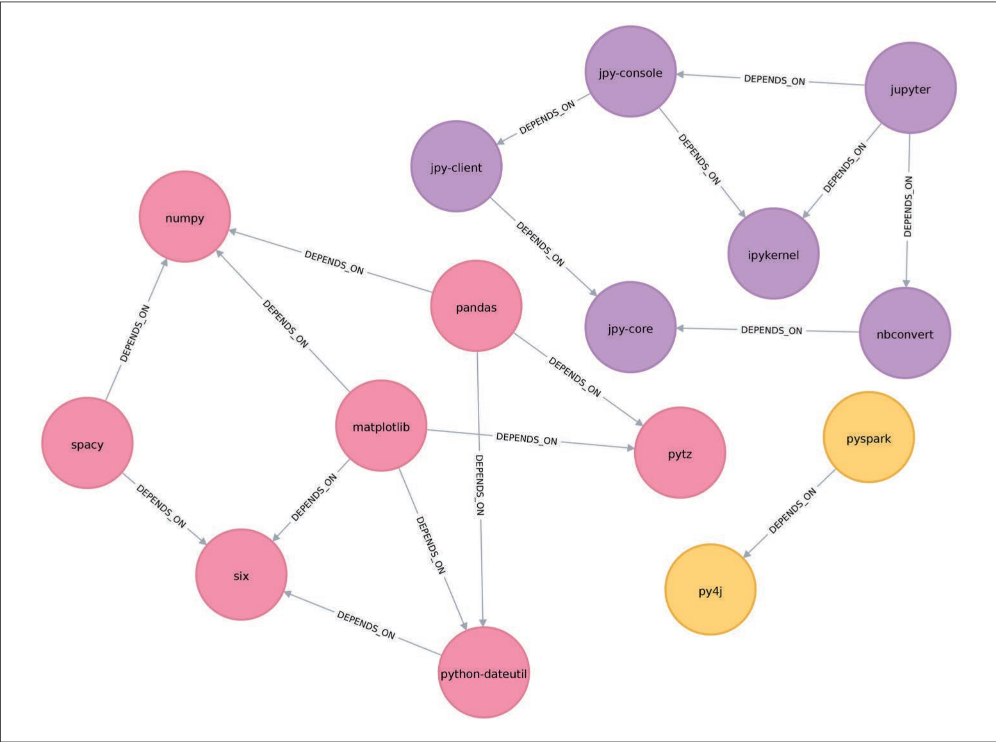


图 6-7：通过连通分量算法发现的簇

6.4.3 使用Neo4j实现连通分量算法

要使用 Neo4j 运行连通分量算法，请执行以下查询。

```
CALL algo.unionFind.stream("Library", "DEPENDS_ON")
YIELD nodeId,setId
RETURN setId, collect(algo.getNodeById(nodeId)) AS libraries
ORDER BY size(libraries) DESC
```

传递给该算法的参数如下。

- Library
  从图加载的节点标签。
- DEPENDS_ON
  从图加载的关系类型。

输出结果如下所示：

setId	libraries
2	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
5	[jupyter, jpy-console, nbconvert, ipykernel, jpy-client, jpy-core]
1	[pyspark, py4j]

不出所料，结果与 Spark 实现的完全相同。

前面介绍的两种社团发现算法都属于确定性算法，即每次运行都返回相同的结果。接下来介绍两种不确定性算法，多次运行的话，即使数据相同，结果也可能不同。

6.5 标签传播算法

标签传播算法（label propagation algorithm，LPA）是在图中查找社团的一种快速算法。在该算法中，节点基于直接邻节点选择群组。这种方法非常适用于分组不那么清晰、可以用权重辅助确定节点所属社团的网络。标签传播算法也很适合半监督学习，这是因为可以使用预先分配的指示性节点标签作为种子。

该算法的思想是，一个标签可以很快主导连接稠密的节点群组，但它在跨越连接稀疏的区域时会遇到困难。当算法完成时，标签被困在连接稠密的节点群组中，最终具有相同标签的节点可视为在同一社团中。该算法将成员关系指派给邻域内关系和节点组合权重最高的标签，这样就解决了重叠问题（节点可能属于多个簇）。标签传播算法是 2007 年由 Usha Nandini Raghavan、Reka Albert 和 Soundar Kumara 在论文“Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks”中提出的一种相对较新的算法。

图 6-8 展示了标签传播算法的两种变体，即较为简单的推方法和更为典型的拉方法（依赖关系权重）。拉方法很适合并行化处理。

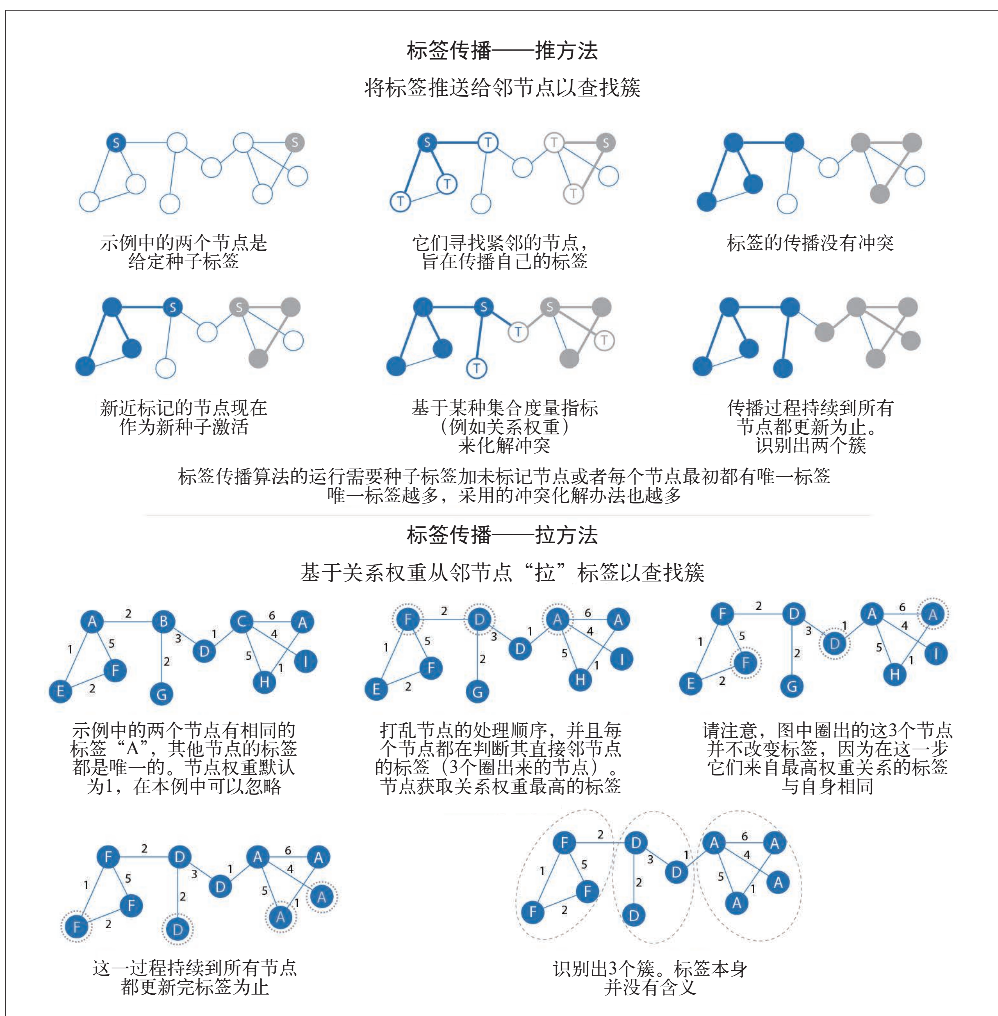


图 6-8：标签传播算法的两种变体

标签传播拉方法的常用步骤如下。

1. 每个节点都用唯一标签（一个标识符）初始化，而且可以选用初始“种子”标签。
2. 这些标签通过网络传播。
3. 在每次传播迭代中，每个节点都会更新其标签以匹配具有最大权重的标签，最大权重是根据邻节点的权重及其关系计算的。联系是均匀而随机断开的。
4. 当每个节点都拥有大多数邻节点的标签时，标签传播算法收敛。

随着标签的传播，连接稠密的节点群组很快会就某个唯一标签达成一致。在传播结束时只剩下几个标签，而具有相同标签的节点属于同一社团。

6.5.1 半监督学习和种子标签

与其他算法不同，标签传播算法可以在同一图上多次运行并返回不同的社团结构。节点的评估顺序对最终返回的社团有影响。

当某些节点被赋予初始标签（种子标签），而其他节点没有标记时，求解范围会变窄。未标记的节点更可能采用初始标签。

可将使用标签传播算法视为一种针对社团查找的**半监督学习方法**。半监督学习是一类机器学习任务和技术，针对少量有标记的数据和大量无标记的数据进行运算。当图发生演变时，还可以在图上重复运行算法。

此外，标签传播算法有时无法收敛于单个解。在这种情况下，社团发现的结果将在几个非常相似的社团之间持续摆荡，而且算法永远无法完成。种子标签有助于引导算法得出解。Spark 和 Neo4j 使用一个最大迭代次数集合来避免执行过程无休止。你应该针对自己的数据来测试迭代设置，以平衡准确率和执行时间。

6.5.2 何时使用标签传播算法

标签传播算法通常用于发现大规模网络中的初始社团，当权重可用时尤其适用。该算法可以并行化处理，因此在分割图时速度非常快。

示范用例如下。

- 将推文的“极性”作为语义分析的一部分。在该场景中，将来自分类器的“正负”种子标签与 Twitter 关注者图结合使用。
- 根据化学相似性和副作用概况，找出同开的药物中是否存在潜在危险组合，参见 Ping Zhang 等人的论文“Label Propagation Prediction of Drug-Drug Interactions Based on Clinical Side Effects”。
- 针对机器学习模型推断对话特征和用户意图。

6.5.3 使用Spark实现标签传播算法

首先启动 Spark，从 Spark 和 GraphFrames 包中导入所需的包：

```
from pyspark.sql import functions as F
```

然后准备运行标签传播算法，代码如下所示：

```
result = g.labelPropagation(maxIter=10)
(result
.sort("label")
.groupby("label")
.agg(F.collect_list("id"))
.show(truncate=False))
```

在 PySpark 中运行这段代码，输出结果如下所示：

label	collect_list(id)
180388626432	[jpy-core, jpy-console, jupyter]
223338299392	[matplotlib, spacy]
498216206336	[python-dateutil, numpy, six, pytz]
549755813888	[pandas]
558345748480	[nbconvert, ipykernel, jpy-client]
936302870528	[pyspark]
1279900254208	[py4j]

与连通分量算法相比，本例中库的簇更多。在如何判定簇方面，标签传播算法没有连通分量算法那么严格。使用标签传播算法，两个相邻（直接连接）的节点可能位于不同的簇中；然而，在使用连通分量算法时，节点总是与其邻节点位于同一个簇中，这是因为该算法严格基于关系进行分组。

在本例中，最明显的区别是 Jupyter 的库被划为两个社团，一个包含其核心部分，另一个为面向客户端的工具。

6.5.4 使用Neo4j实现标签传播算法

下面使用 Neo4j 来尝试同一算法。执行以下查询：

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
  { iterations: 10 })
YIELD nodeId, label
RETURN label,
       collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

传递给该算法的参数如下。

- `Library`

  从图加载的节点标签。

- `DEPENDS_ON`

  从图加载的关系类型。

- `iterations: 10`

  运行时的最大迭代次数。

结果如下所示：

label	libraries
11	[matplotlib, spacy, six, pandas, python-dateutil]
10	[jupyter, jpy-console, nbconvert, jpy-client, jpy-core]
4	[pyspark, py4j]
8	[ipykernel]
13	[numpy]
0	[pytz]

图 6-9 将上述结果可视化，可以看到，这与用 Spark 得到的结果相似。

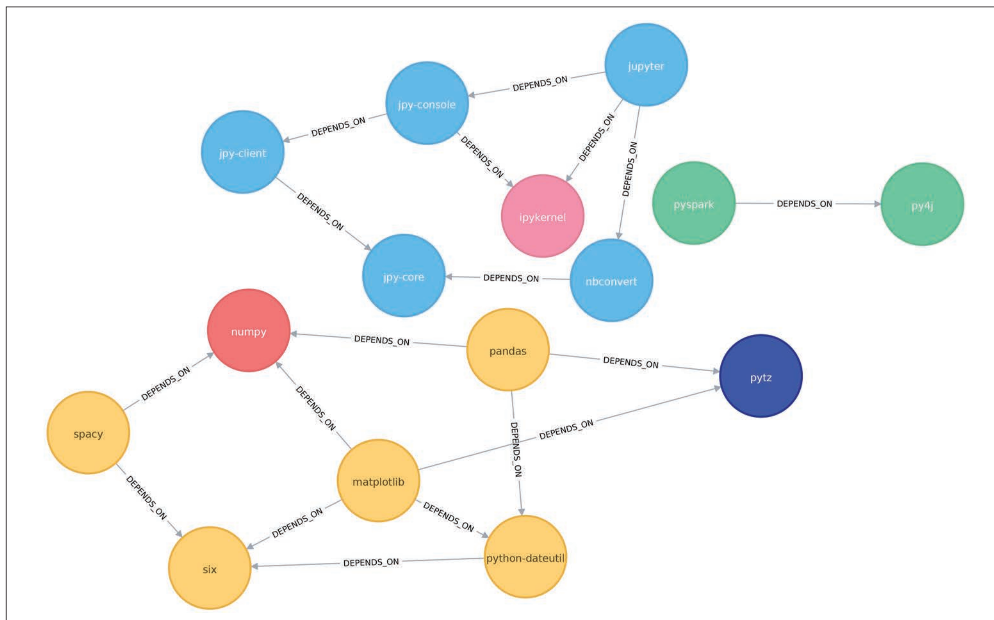


图 6-9：通过标签传播算法查找簇

还可以在假设图无向的情况下运行该算法，这意味着节点将尝试采用它们所依赖的库的标签和依赖它们的库的标签。

为此，将算法的 direction 参数设置为 BOTH：

```
CALL algo.labelPropagation.stream("Library", "DEPENDS_ON",
  { iterations: 10, direction: "BOTH" })
YIELD nodeId, label
RETURN label,
       collect(algo.getNodeById(nodeId).id) AS libraries
ORDER BY size(libraries) DESC
```

运行以上代码，输出结果如下所示：

label	libraries
11	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
10	[nbconvert, jpy-client, jpy-core]
6	[jupyter, jpy-console, ipykernel]
4	[pyspark, py4j]

簇的数量从 6 减少到 4，现在图中 matplotlib 部分的所有节点都被分到一组了，如图 6-10 所示。



图 6-10：当忽略关系方向时，通过标签传播算法发现簇

虽然在该数据之上运行标签传播算法时，无向图和有向图的计算结果相似，但是在复杂图上，差异会更显著。这是因为忽略方向会导致节点尝试采用更多标签，而不考虑关系的来源。

6.6 Louvain模块度算法

Louvain 模块度算法在将节点分配到不同群组时，通过比较社团密度来查找簇。可以将其视为一种假设分析，尝试多种分组方式以达到全局最优。

Louvain 模块度算法诞生于 2008 年，是最快的一种基于模块度的算法。除发现社团外，它还揭示了不同尺度上的社团层级结构，这对于理解不同粒度的网络结构非常有用。

通过将簇内连接的密度与平均值或随机样本进行比较，Louvain 模块度算法可以量化评估将某个节点分配给某个群组是否恰当。这种度量社团分配的指标称为**模块度**。

6.6.1 通过模块度进行基于质量的分组

模块度是一种发现社团的技术，它将图分割为更粗粒度的模块（或簇），然后度量本次分组的强度。与仅仅查看簇内连接的集中度不同，该方法将簇内关系密度与簇间关系密度进行比较。模块度是度量这些分组质量的指标。

模块度算法首先实现社团的局部优化，然后再实现全局优化，使用多组迭代来测试不同分组方法并且增加粗粒度。这种策略可以识别社团的层级结构，比较全面地认识整体结构。然而，所有模块度算法都存在以下两个缺点：

- 它们将较小的社团合并成较大的社团；
- 当多个分割选项出现相似的模块度时，可能会出现停滞，形成局部极点并阻碍进一步处理。

更多相关信息，请参阅 Benjamin H. Good、Yves-Alexandre de Montjoye 和 Aaron Clauset 的论文“Performance of Modularity Maximization in Practical Contexts”。

Louvain 模块度算法首先对所有节点的模块度进行局部优化，找到若干小社团；然后将每个小社团分组到更大的联合集团，之后重复第一步，直到达到全局最优为止。

计算模块度

一种简单的模块度计算方法是用给定群组内关系的分值减去关系在所有节点间随机分布的预期值。该值总是介于 –1 ～ 1，正值表示关系密度大于预期值，负值表示关系密度小于预期值。图 6-11 展示了基于节点分组的几个模块度得分。

群组的模块度公式如下：

$$M = \sum_{c=1}^{n_c} \left[\frac{L_c}{L} - \left(\frac{k_c}{2L} \right)^2 \right]$$

其中：

- L 是整个群组的关系数；
- L_c 是某个分割中的关系数；
- k_c 是分割中节点的总度数。

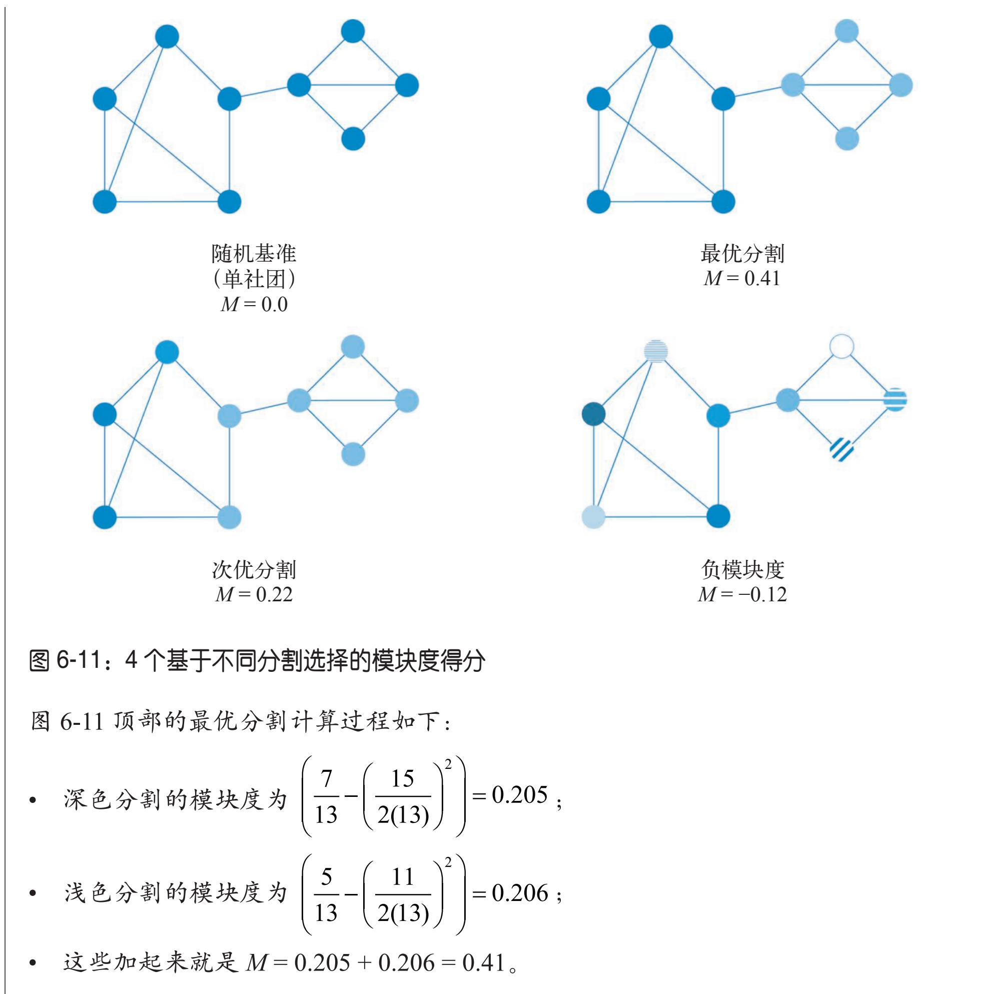


图 6-11：4 个基于不同分割选择的模块度得分

图 6-11 顶部的最优分割计算过程如下：

- 深色分割的模块度为 $\left(\frac{7}{13}-\left(\frac{15}{2(13)}\right)^2\right)=0.205$；
- 浅色分割的模块度为 $\left(\frac{5}{13}-\left(\frac{11}{2(13)}\right)^2\right)=0.206$；
- 这些加起来就是 $M = 0.205 + 0.206 = 0.41$。

该算法包含两个要重复应用的步骤，如图 6-12 所示。

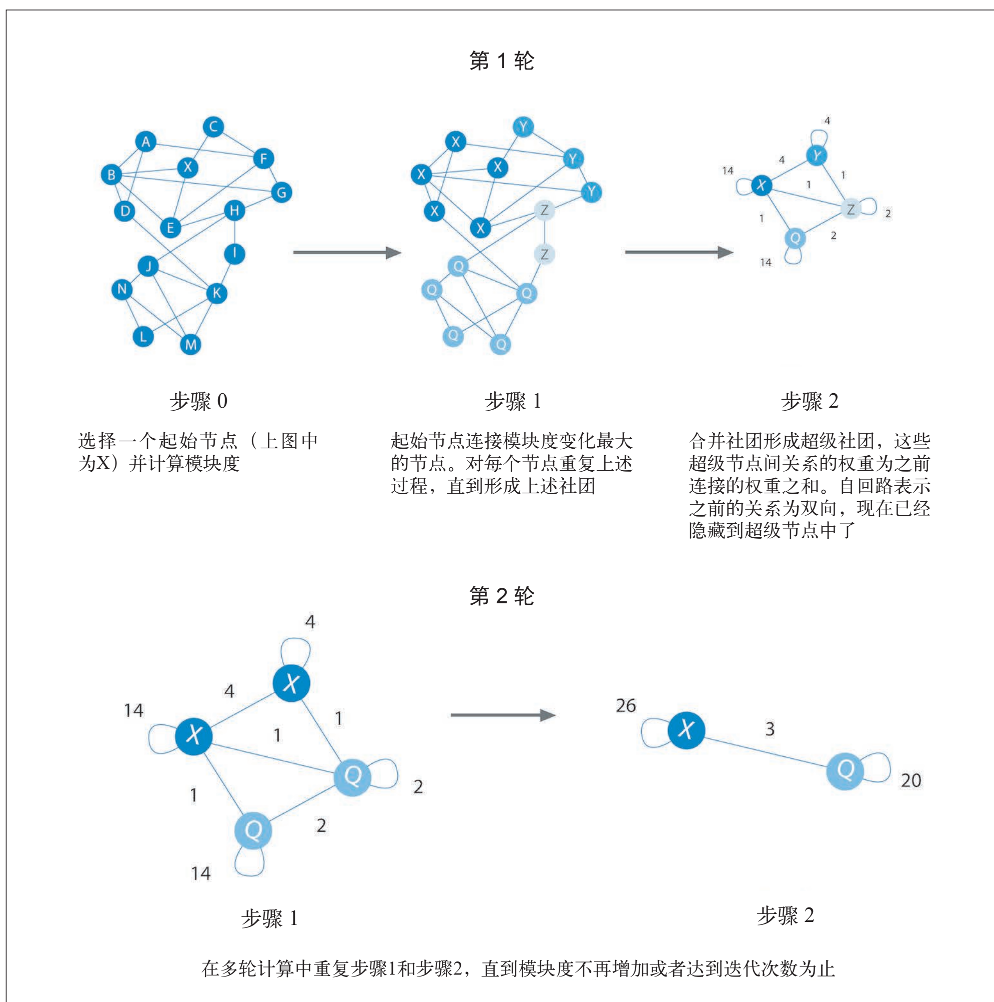


图 6-12：Louvain 模块度算法的过程

Louvain 算法的步骤如下。

1. 将节点“贪婪”地分配给社团，侧重于模块度的局部优化。
2. 根据第 1 步中找到的社团定义更粗粒度的网络，这种网络将在算法的下一次迭代中使用。

重复这两个步骤，直到社团的模块度无法继续增加为止。

对第 1 步的优化操作之一是评估群组的模块度。Louvain 模块度算法用以下公式来实现。

$$Q = \frac{1}{2m}\sum_{u,v}\left[A_{uv} - \frac{k_u k_v}{2m}\right]\delta(c_u, c_v)$$

解释如下。

- u 和 v 都是节点。
- m 是整个图的关系总权重（在模块度公式中，$2m$ 是归一化的常用取值）。
- $A_{uv} - \frac{k_u k_v}{2m}$ 是 u 和 v 间关系强度与将网络各节点随机分配的预期值（趋于平均值）的比较值。
  - A_{uv} 是 u 和 v 间关系的权重。
  - k_u 是 u 的关系权重和。
  - k_v 是 v 的关系权重和。
- 如果 u 和 v 被分配到同一社团中，则 $\delta(c_u, c_v)$ 的值为 1；如果未在同一社团中，则为 0。

对第 1 步的另一优化操作是，当把节点移动到另一个群组时，评估模块度的变化情况。Louvain 模块度算法使用了该公式的一个更复杂的变体，然后确定最佳群组分配情况。

6.6.2 何时使用Louvain模块度算法

Louvain 模块度算法可用于在大型网络中发现社团。因为计算开销大，所以 Louvain 模块度算法并没有完全采用模块度，而是采用了一个启发式函数。因此，Louvain 模块度算法可用于处理标准模块度算法难以处理的大型图。

Louvain 模块度算法对于评估复杂网络的结构也很有帮助，尤其擅长发现多层级结构，比如可能会发现一个犯罪组织的层级结构。该算法还可以在不同粒度上缩放，在子社团的子社团内部查找子社团。

示范用例如下。

- 检测网络攻击。Sunanda Vivek Shanbhaq 在 2016 年的一项研究中使用了 Louvain 模块度算法，面向网络安全应用探索大规模网络的快速社团检测。一旦发现这些社团，就可以用它们来检测网络攻击。
- 将文档中共同出现的词汇引入主题建模过程，并基于此从在线社交平台上提取主题。
- 发现大脑功能网络中的层级社团结构，参见 David Meunier 等人的论文“Hierarchical Modularity in Human Brain Functional Networks”。



包括 Louvain 模块度算法在内的模块度优化算法都存在两个问题。首先，这些算法忽略了大型网络中的小型社团。可以通过审查中间合并步骤来解决这个问题。其次，在具有重叠社团的大型图中，模块度优化器可能无法正确判定全局极值。对于后一种情况，推荐使用任意一种模块度算法作为指引，以进行估计（不完全准确）。

6.6.3 使用Neo4j实现Louvain模块度算法

下面看看如何运行 Louvain 模块度算法。可以执行以下查询：

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
RETURN algo.getNodeById(nodeId).id AS libraries, communities
```

传递给该算法的参数如下。

- `Library`

  从图加载的节点标签。

- `DEPENDS_ON`

  从图加载的关系类型。

结果如下所示：

libraries	communities
pytz	[0, 0]
pyspark	[1, 1]
matplotlib	[2, 0]
spacy	[2, 0]
py4j	[1, 1]
jupyter	[3, 2]
jpy-console	[3, 2]
nbconvert	[4, 2]
ipykernel	[3, 2]
jpy-client	[4, 2]
jpy-core	[4, 2]
six	[2, 0]
pandas	[0, 0]
numpy	[2, 0]
python-dateutil	[2, 0]

`communities` 列描述了由可分为两个层级的节点构成的社团。数组中前一个值标记中间社团，后一个值标记最终社团。

分配给中间社团和最终社团的数字只是没有度量意义的标签，可理解为用于指示节点所属社团的标签，如“属于标记为 0 的社团”“标记为 4 的社团”等。

举例来说，matplotlib 的结果是 [2,0]，这意味着 matplotlib 的最终社团被标记为 0，而它的中间社团被标记为 2。

如果先使用算法的写版本存储这些社团，再进行查询，就更容易看出该算法的工作流程。下述查询将运行 Louvain 模块度算法，并将结果存储在每个节点的 communities 性质中：

```
CALL algo.louvain("Library", "DEPENDS_ON")
```

还可以使用该算法的流版本来存储产生的社团，之后调用 SET 子句来存储结果。利用下面的查询可实现：

```
CALL algo.louvain.stream("Library", "DEPENDS_ON")
YIELD nodeId, communities
WITH algo.getNodeById(nodeId) AS node, communities
SET node.communities = communities
```

在执行这两个查询后，还可以编写以下查询来查找最终簇。

```
MATCH (l:Library)
RETURN l.communities[-1] AS community, collect(l.id) AS libraries
ORDER BY size(libraries) DESC
```

l.communities[-1] 返回了该性质存储于底层数组中的最后一项。

执行该查询，结果如下所示：

community	libraries
0	[pytz, matplotlib, spacy, six, pandas, numpy, python-dateutil]
2	[jupyter, jpy-console, nbconvert, ipykernel, jpy-client, jpy-core]
1	[pyspark, py4j]

该聚类结果与连通分量算法的结果一样。

matplotlib 位于包含 pytz、spacy、six、pandas、numpy 和 python-dateutil 的社团中，如图 6-13 所示。



图 6-13：由 Louvain 模块度算法发现的簇

Louvain 模块度算法的另一个特点是可以看到中间聚类结果，这可以展示比最终层粒度更细的簇：

```
MATCH (l:Library)
RETURN l.communities[0] AS community, collect(l.id) AS libraries
ORDER BY size(libraries) DESC
```

执行该查询，结果如下所示：

community	libraries
2	[matplotlib, spacy, six, python-dateutil]
4	[nbconvert, jpy-client, jpy-core]
3	[jupyter, jpy-console, ipykernel]
1	[pyspark, py4j]
0	[pytz, pandas]
5	[numpy]

matplotlib 所在社团的库现在分成了 3 个较小的社团：

- matplotlib、spacy、six 和 python-dateutil；
- pytz 和 pandas；
- numpy。

图 6-14 直观地展现了这一细分结果。

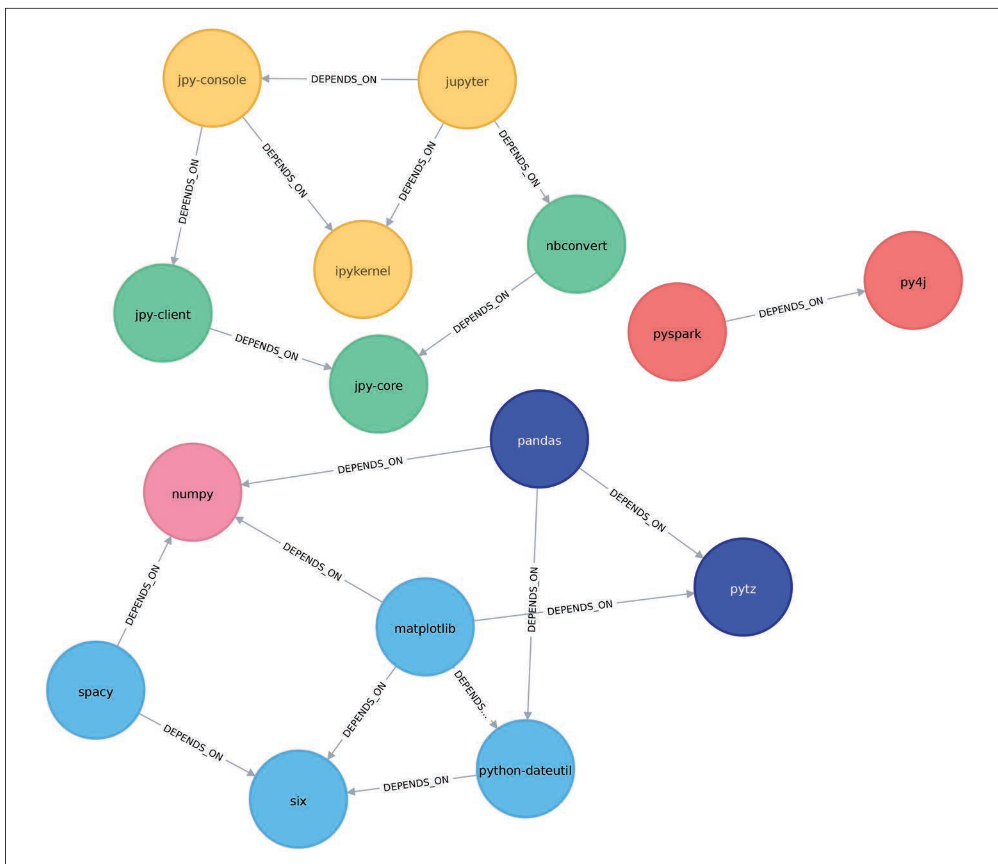


图 6-14：通过 Louvain 模块度算法查找中间簇

虽然该图显示的层级结构只包含两层，但是如果在更大的图上运行该算法，就会看到更复杂的层级结构。Louvain 模块度算法揭示的中间簇对于发现细粒度的社团非常有用，而其他社团发现算法可能无法发现这些细粒度的社团。

6.7　验证社团

社团发现算法通常具有共同的目标，即识别群组。然而，由于不同的算法基于不同的假设，因此它们可能会发现不同的社团。这使得为特定问题选择正确算法更具挑战性，还带有一点探索的意味。

当与周围环境相比群组内部的关系较密时，大多数社团发现算法表现得相当不错，但在现实世界的网络中，这一特征并不明显。通过将算法结果与已知社团数据的基准进行比较，可以验证社团发现的准确性。

最著名的两个基准是 Girvan-Newman（GN）算法和 Lancichinetti-Fortunato-Radicchi（LFR）算法。这两个算法生成的参考网络极为不同：GN 生成一个较为同构的随机网络，而 LFR 创建一个较为异构的图，其中节点的度和社团规模按照幂律分布。

由于测试的准确性取决于所使用的基准，因此基准与数据集的匹配非常重要。应尽可能寻找密度、关系分布、社团定义和相关领域相似的基准。

6.8　小结

社团发现算法有助于理解对图中节点进行分组的方式。

本章首先介绍了三角形计数和聚类系数，然后讨论了两种具有确定性的社团发现算法：强连通分量算法和连通分量算法。这些算法对社团的构成有严格定义，对于在图分析流程的早期了解图结构非常有用。

本章还介绍了标签传播算法和 Louvain 模块度算法，这两种不确定性算法能够更好地发现较细粒度的社团。Louvain 模块度算法能在不同尺度上展示社团的层级。

第 7 章将使用规模更大的数据集，介绍如何将这些算法组合使用，进而洞察关联数据。

第7章 图算法实战

随着对各种算法在特定数据集上的表现越来越熟悉，我们所采用的图分析方法也在不断深入。本章将介绍几个示例，展示如何针对 Yelp 数据集和美国交通数据集做大规模图数据分析。我们将在 Neo4j 平台上分析 Yelp 数据，包括概览数据，组合算法实现行程推荐，以及挖掘用户数据和业务数据以供咨询。我们将在 Spark 中研究美国航空公司数据，了解交通模式、航班延误以及各航空公司与机场的关联方式。

由于路径查找算法相对简单，因此示例主要采用下述中心性算法和社团发现算法。

- 用 PageRank 算法查找有影响力的 Yelp 评论者，然后关联他们对特定酒店的评级。
- 用中间中心性算法发现连接多个群组的评论者，然后获取他们的偏好。
- 用标签传播算法以及投影操作为相似的 Yelp 业务创建超类别。
- 用度中心性算法在美国交通数据集中快速识别机场枢纽。
- 用强连通分量算法研究美国机场的航线簇。

7.1 使用Neo4j分析Yelp数据

Yelp 根据评论、偏好和推荐帮助人们查找当地商家。截至 2018 年底，用户在该平台上已发表 1.8 亿多条评论。Yelp 自 2013 年起举办了多场 Yelp 数据集挑战赛，鼓励人们探索和研究 Yelp 的开放数据集。

第 12 届 Yelp 数据集挑战赛于 2018 年举办，其开放数据集包含以下内容。

- 700 多万条评论和提示。
- 150 多万用户和 28 万张照片。

- 超过 18.8 万户商家，涉及 140 万个属性。
- 覆盖 10 个大都市地区。

该数据集自发布以来广受欢迎，有数百篇学术论文使用了其中的数据。Yelp 数据集代表了具有良好数据结构且高度互联的真实数据。该数据集很适于展示图算法，你可以下载该数据集并进行研究。

7.1.1 Yelp社交网络

除了撰写和阅读关于商家的评论，Yelp 用户还形成了一个社交网络。用户在浏览 Yelp 官网时可以向遇到的其他用户发送好友请求，也可以关联其通讯录或 Facebook 图。

Yelp 数据集也含有社交网络。图 7-1 是 Mark 的 Yelp 个人资料中 Friends 部分的屏幕截图。

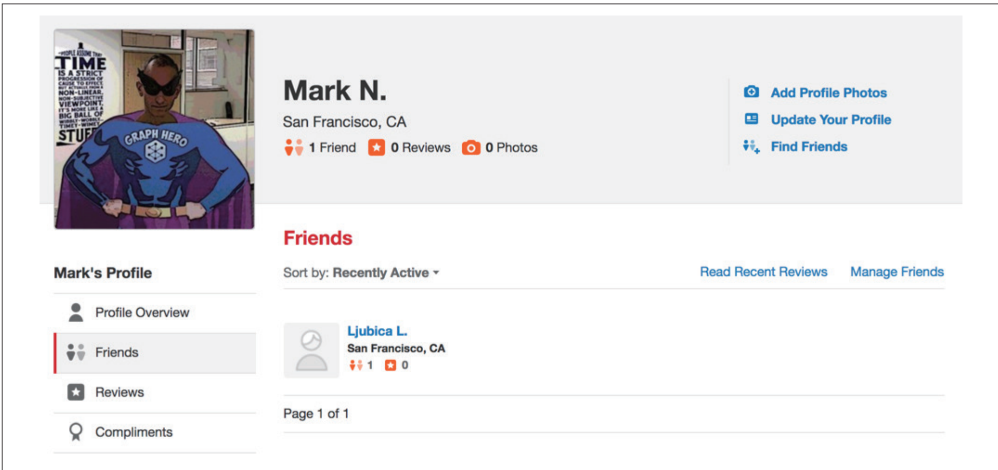


图 7-1：Mark 的 Yelp 个人资料

除了需要给 Mark 多添加几个朋友之外，其他一切都已经准备就绪了。为了说明如何在 Neo4j 中分析 Yelp 数据，这里考虑旅游信息提供商的工作场景。首先探查 Yelp 数据，然后看看如何用应用程序帮助人们计划行程。本章将逐步介绍如何在像拉斯维加斯这样的大城市里查找关于住宿和游玩的建议。

另一部分业务场景是为旅游目的地的商家提供咨询。下面通过示例说明如何帮助酒店识别有影响力的客人，然后帮其确定应该针对哪些业务开展促销活动。

7.1.2 导入数据

将数据导入 Neo4j 有多种方法，包括使用导入工具、采用 `LOAD CSV` 命令（之前讲过）和 Neo4j 驱动程序等。

对于 Yelp 数据集而言，由于需要一次性导入大量数据，因此导入工具是最佳选择，详见附录。

7.1.3 图模型

Yelp 数据可用图 7-2 所示的图模型表示。

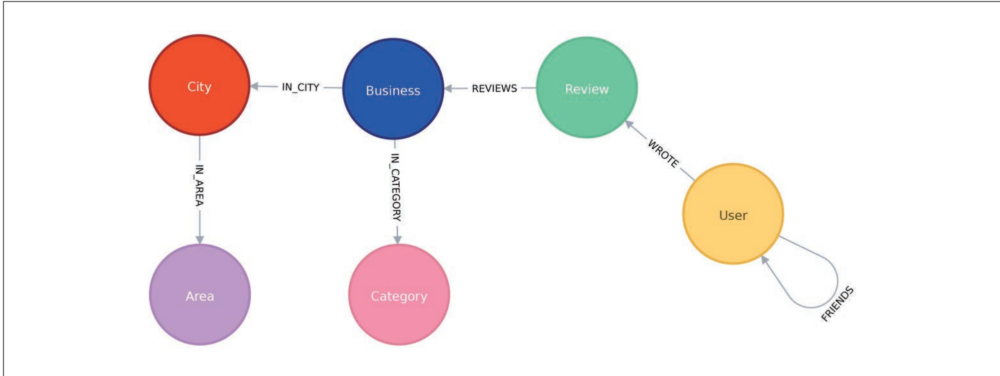


图 7-2：Yelp 图模型

该图包含带有 `User` 标记的节点，该节点与其他 `User` 有 `FRIENDS` 关系。`User` 还会撰写关于 `Business` 的 `Review` 和提示。所有元数据都以节点属性存储，而业务类别除外，单独用 `Category` 节点表示。将属性 `City` 和 `Area` 抽取到子图中，可得到位置数据。在其他用例中，将日期等其他属性抽取到节点或将节点折叠到关系（如评论）也很有意义。

Yelp 数据集还包括用户提示和照片，但本例不会用到这些信息。

7.1.4 Yelp数据概览

将数据加载到 Neo4j 中后，就可以执行一些探索性查询了。为了了解 Yelp 数据，要探查每个类别中的节点数量和存在的关系类型。之前展示了针对 Neo4j 示例的 Cypher 查询，不过还可以用另一种编程语言来执行这些查询。由于 Python 是数据科学家的首选语言，因此在想把结果关联到 Python 生态系统中的其他库时，可使用 Neo4j 的 Python 驱动程序。如果只想显示查询结果，则可直接使用 Cypher。

本章将展示如何把 Neo4j 与流行的 pandas 库结合使用，该库可有效整理数据库外部的数据。本章还将介绍如何使用 tabulate 库来美化从 pandas 获得的结果，以及如何使用 matplotlib 库将数据可视化。

此外，还可以使用 Neo4j 的 APOC 程序库来辅助编写功能更强大的 Cypher 查询。附录将介绍关于 APOC 的更多内容。

首先安装 Python 库：

```
pip install neo4j-driver tabulate pandas matplotlib
```

然后，导入以下库：

```
from neo4j.v1 import GraphDatabase
import pandas as pd
from tabulate import tabulate
```

在 macOS 上导入 matplotlib 库有些烦琐，但是用以下代码应该能够成功：

```
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

如果在其他操作系统上运行，可能不需要中间一行代码。下面创建一个指向本地 Neo4j 数据库的 Neo4j 驱动程序实例。

```
driver = GraphDatabase.driver("bolt://localhost", auth=("neo4j", "neo"))
```



要使用自己的主机和证书，还需要更新驱动程序的初始值。

首先研究关于节点和关系的一些常规数值。以下代码计算数据库中节点标签的基数（计算每个标签对应的节点数）：

```
result = {"label": [], "count": []}
with driver.session() as session:
    labels = [row["label"] for row in session.run("CALL db.labels()")]
    for label in labels:
        query = f"MATCH (:`{label}`) RETURN count(*) as count"
        count = session.run(query).single()["count"]
        result["label"].append(label)
        result["count"].append(count)

df = pd.DataFrame(data=result)
print(tabulate(df.sort_values("count"), headers='keys',
                              tablefmt='psql', showindex=False))
```

运行以上代码，可以看到每个标签的节点数量，如下所示：

label	count
Area	54
City	1093
Category	1293

Business	174567
User	1326101
Review	5261669

可以通过以下代码来将基数可视化：

```
plt.style.use('fivethirtyeight')

ax = df.plot(kind='bar', x='label', y='count', legend=None)

ax.xaxis.set_label_text("")
plt.yscale("log")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

以上代码生成的图表如图 7-3 所示。注意，该图表采用了对数刻度。

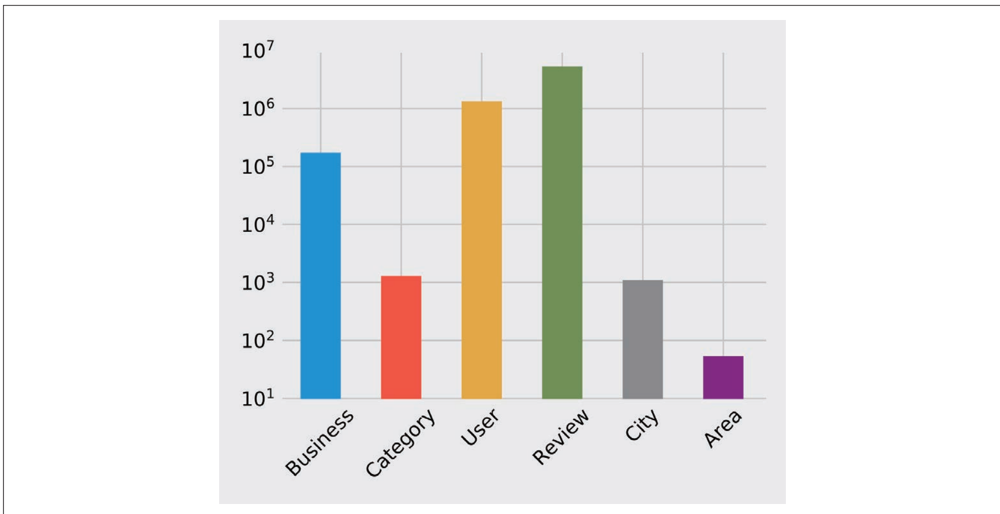


图 7-3：每个标签类别下的节点数量

同理，还可以计算关系的基数：

```
result = {"relType": [], "count": []}
with driver.session() as session:
    rel_types = [row["relationshipType"] for row in session.run
                    ("CALL db.relationshipTypes()")]
    for rel_type in rel_types:
        query = f"MATCH ()-[:`{rel_type}`]->() RETURN count(*) as count"
        count = session.run(query).single()["count"]
        result["relType"].append(rel_type)
        result["count"].append(count)
```

```
df = pd.DataFrame(data=result)
print(tabulate(df.sort_values("count"), headers='keys',
                              tablefmt='psql', showindex=False))
```

运行这段代码，可以得到每种关系的数量，如下所示：

relType	count
IN_AREA	1154
IN_CITY	174566
IN_CATEGORY	667527
WROTE	5261669
REVIEWS	5261669
FRIENDS	10645356

以上基数的图表如图 7-4 所示。和节点基数图表一样，该图表也使用了对数刻度。

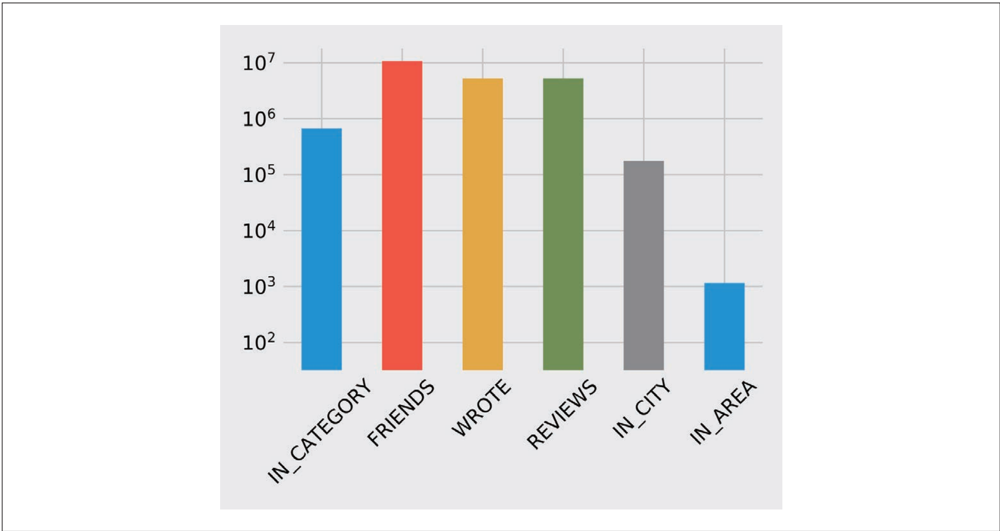


图 7-4：按关系类型划分的关系数量

这些查询并没有揭示任何惊人的结果，但有助于了解数据的内容。还可以通过这些查询结果快速检查导入的数据是否正确。

假设 Yelp 有很多关于酒店的评论，但是在关注评论区之前有必要先进行检查。执行以下查询，可了解数据中酒店和评论的数量。

```
MATCH (category:Category {name: "Hotels"})
RETURN size((category)<-[:IN_CATEGORY]-()) AS businesses,
       size((:Review)-[:REVIEWS]->(:Business)-[:IN_CATEGORY]->
                                   (category)) AS reviews
```

结果如下所示：

businesses	reviews
2683	183759

确实有很多酒店和评论。接下来在业务场景中进一步研究数据。

7.1.5 行程规划应用程序

为了在应用程序中增加受欢迎的推荐功能，首先要找到评级最高的酒店，以供热门预订的启发式函数使用。可以添加其评级信息以了解实际体验。使用以下代码查看评论最多的前 10 家酒店并绘制评级分布图：

```
# 查找评论最多的前10家酒店
query = """
MATCH (review:Review)-[:REVIEWS]->(business:Business),
      (business)-[:IN_CATEGORY]->(category:Category {name: $category}),
      (business)-[:IN_CITY]->(:City {name: $city})
RETURN business.name AS business, collect(review.stars) AS allReviews
ORDER BY size(allReviews) DESC
LIMIT 10
"""

fig = plt.figure()
fig.set_size_inches(10.5, 14.5)
fig.subplots_adjust(hspace=0.4, wspace=0.4)

with driver.session() as session:
    params = { "city": "Las Vegas", "category": "Hotels"}
    result = session.run(query, params)
    for index, row in enumerate(result):
        business = row["business"]
        stars = pd.Series(row["allReviews"])

        total = stars.count()
        average_stars = stars.mean().round(2)

        # 计算星级分布
        stars_histogram = stars.value_counts().sort_index()
        stars_histogram /= float(stars_histogram.sum())

        # 绘制直方图，以展示星级分布情况
        ax = fig.add_subplot(5, 2, index+1)
        stars_histogram.plot(kind="bar", legend=None, color="darkblue",
                             title=f"{business}\nAve:
                                     {average_stars}, Total: {total}")

plt.tight_layout()
plt.show()
```

我们限定城市和类别，仅关注拉斯维加斯（Las Vegas）的酒店。运行以上代码，可得到图 7-5 所示的图表。请注意，x 轴表示酒店的评级，y 轴表示各评级所占的百分比（由小数表示）。

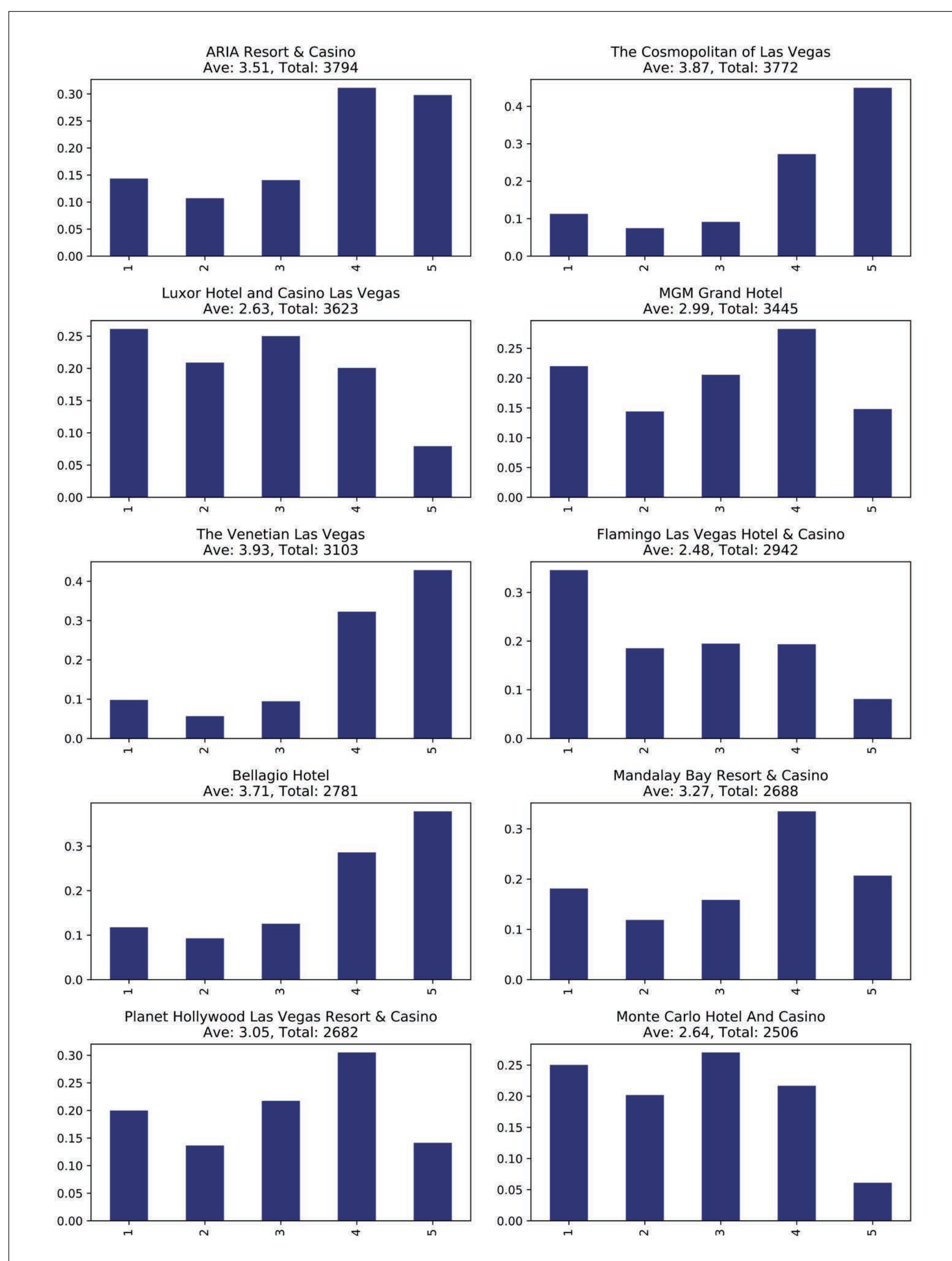
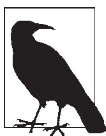


图 7-5：评论数最多的前 10 家酒店，*x* 轴表示评级，*y* 轴为全部评级所占百分比

人们不可能阅读全部的酒店评论。更好的做法是仅给出与用户相关的评论，并在应用程序中突出显示。要进行这样的分析，就要从基本的探查转向使用图算法。

查找有影响力的酒店评论者

要判断应该展示哪些评论，方法之一就是根据 Yelp 上评论者的影响力对评论进行排序。对所有至少浏览过 3 家酒店的用户进行投影，在生成的图上运行 PageRank 算法。如前所述，投影操作有助于过滤不必要的信息，还可以添加关系数据（有时是推断出来的）。我们将使用 Yelp 的朋友图（7.1.1 节介绍过）来研究用户之间的关系。PageRank 算法将揭示那些对多数用户有较大影响力的评论者（即使他们并没有直接的朋友关系）。



如果两个人互为 Yelp 朋友，他们之间就有两个 `FRIENDS` 关系。如果 A 和 B 是朋友，那么从 A 到 B 就有 `FRIENDS` 关系，从 B 到 A 也有 `FRIENDS` 关系。

需要写一个查询把发表过 3 条以上评论的用户投影到子图上，然后在该投影子图上运行 PageRank 算法。

举例说明子图投影的工作流程。图 7-6 显示了 3 个互为朋友的用户 Mark、Arya 和 Praveena 之间的关系图。Mark 和 Praveena 都评价了 3 家酒店，这将成为投影图的一部分；而 Arya 只评论过一家酒店，因此将被排除在投影图之外。



图 7-6：样例 Yelp 图

投影图仅包含 Mark 和 Praveena，如图 7-7 所示。

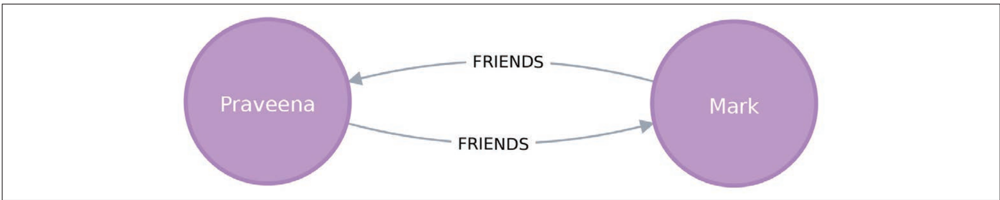


图 7-7：样例投影图

图投影的工作原理已介绍完毕。下面的查询在投影图上运行 PageRank 算法，并将结果存储在每个节点的 `hotelPageRank` 属性中。

```
CALL algo.pageRank(
  'MATCH (u:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
                              (:Category {name: $category})
   WITH u, count(*) AS reviews
   WHERE reviews >= $cutOff
   RETURN id(u) AS id',
  'MATCH (u1:User)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
                              (:Category {name: $category})
   MATCH (u1)-[:FRIENDS]->(u2)
   RETURN id(u1) AS source, id(u2) AS target',
  {graph: "cypher", write: true, writeProperty: "hotelPageRank",
   params: {category: "Hotels", cutOff: 3}}
)
```

可以看到，这里并没有像第 5 章那样设置阻尼系数或最大迭代限制。如果没有显式设置，Neo4j 默认将阻尼系数设为 0.85，将 `maxIterations` 设为 20。

下面看看 PageRank 值的分布情况，以及如何筛选数据。

```
MATCH (u:User)
WHERE exists(u.hotelPageRank)
RETURN count(u.hotelPageRank) AS count,
       avg(u.hotelPageRank) AS ave,
       percentileDisc(u.hotelPageRank, 0.5) AS `50%`,
       percentileDisc(u.hotelPageRank, 0.75) AS `75%`,
       percentileDisc(u.hotelPageRank, 0.90) AS `90%`,
       percentileDisc(u.hotelPageRank, 0.95) AS `95%`,
       percentileDisc(u.hotelPageRank, 0.99) AS `99%`,
       percentileDisc(u.hotelPageRank, 0.999) AS `99.9%`,
       percentileDisc(u.hotelPageRank, 0.9999) AS `99.99%`,
       percentileDisc(u.hotelPageRank, 0.99999) AS `99.999%`,
       percentileDisc(u.hotelPageRank, 1) AS `100%`
```

执行该查询，结果如下所示：

count	ave	50%	75%	90%	95%	99%	99.9%	99.99%	99.999%	100%
1326101	0.1614898	0.15	0.15	0.157497	0.181875	0.330081	1.649511	6.825738	15.27376	22.98046

先解释该百分比表：90% 对应 0.157497，这意味着有 90% 的用户 PageRank 得分较低。99.99% 反映了前 0.01% 评论者的影响力排名，而 100% 是最高的 PageRank 得分。

有趣的是，90% 的用户得分低于 0.16，接近总体平均水平，仅略高于 PageRank 算法初始化时的 0.15。这些数据似乎反映出一种幂律分布——极具影响力的评论者很少。

因为我们只想找到最具影响力的用户，所以编写查询查找 PageRank 得分在前 0.1% 的用户。下面的查询查找 PageRank 得分高于 1.649511 的评论者（注意，这是 99.9% 的群组）：

```
// 仅查找hotelPageRank评分在前0.1%的用户
MATCH (u:User)
WHERE u.hotelPageRank > 1.649511

// 查找这些用户中的前10位
WITH u ORDER BY u.hotelPageRank DESC
LIMIT 10

RETURN u.name AS name,
       u.hotelPageRank AS pageRank,
       size((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
            (:Category {name: "Hotels"})) AS hotelReviews,
       size((u)-[:WROTE]->()) AS totalReviews,
       size((u)-[:FRIENDS]-()) AS friends
```

执行该查询，结果如下所示：

name	pageRank	hotelReviews	totalReviews	friends
Phil	17.361242	15	134	8154
Philip	16.871013	21	620	9634
Carol	12.416060999999997	6	119	6218
Misti	12.239516000000004	19	730	6230
Joseph	12.003887499999998	5	32	6596
Michael	11.460049	13	51	6572
J	11.431505999999997	103	1322	6498
Abby	11.376136999999998	9	82	7922
Erica	10.993773	6	15	7071
Randy	10.748785999999999	21	125	7846

这些结果表明，虽然 Phil 并没有评论过很多酒店，但他是最可信的评论者。他很可能和一些非常有影响力的人有关联，但是如果想要一系列新评论，从个人资料来看，他并非最佳人选。Philip 的得分略低，但是他的朋友最多，而且他写的评论在数量上大约是 Phil 的 5 倍。虽然 J 写的评论最多，也有相当多的朋友，但是 J 的 PageRank 得分并不是最高的，不过仍位列前十。在本应用程序中，我们重点推送 Phil、Philip 和 J 的酒店评论，其中 J 的影响力和评论量综合最优。

通过关联评论改进了应用程序的内置推荐功能，下面转向另一项业务——咨询。

7.1.6 旅游商务咨询

作为咨询服务的一部分，当有影响力的客人写下入住体验后，酒店可订阅消息提醒，以便采取必要措施。首先看看 Bellagio 酒店的评级，按最有影响力的评论者进行排序：

```
query = """\
MATCH (b:Business {name: $hotel})
MATCH (b)<-[:REVIEWS]-(review)<-[:WROTE]-(user)
WHERE exists(user.hotelPageRank)
RETURN user.name AS name,
       user.hotelPageRank AS pageRank,
       review.stars AS stars
"""

with driver.session() as session:
    params = { "hotel": "Bellagio Hotel" }
    df = pd.DataFrame([dict(record) for record in session.run(query, params)])
    df = df.round(2)
    df = df[["name", "pageRank", "stars"]]

top_reviews = df.sort_values(by=["pageRank"], ascending=False).head(10)
print(tabulate(top_reviews, headers='keys', tablefmt='psql', showindex=False))
```

运行以上代码，结果如下所示：

name	pageRank	stars
Misti	12.239516000000004	5
Michael	11.460049	4
J	11.431505999999997	5
Erica	10.993773	4
Christine	10.740770499999998	4
Jeremy	9.576763499999998	5
Connie	9.118103499999998	5
Joyce	7.621449000000001	4
Henry	7.299146	5
Flora	6.7570075	4

请注意，该结果和此前的酒店最佳评论者列表不同，这是因为它只列出了为 Bellagio 酒店评级的评论者。

Bellagio 酒店的客服团队好像不错——影响力前 10 位的评论者都给出了很高的评级。酒店可能希望吸引这些人再次入住并分享体验。

是否有哪位有影响力的客人没有这样好的体验？可以运行以下代码来查找那些 PageRank 得分很高但给出评级低于四星的客人：

```
query = """\
MATCH (b:Business {name: $hotel})
MATCH (b)<-[:REVIEWS]-(review)<-[:WROTE]-(user)
WHERE exists(user.hotelPageRank) AND review.stars < $goodRating
RETURN user.name AS name,
       user.hotelPageRank AS pageRank,
       review.stars AS stars
"""

with driver.session() as session:
    params = { "hotel": "Bellagio Hotel", "goodRating": 4 }
    df = pd.DataFrame([dict(record) for record in session.run(query, params)])
    df = df.round(2)
    df = df[["name", "pageRank", "stars"]]

top_reviews = df.sort_values(by=["pageRank"], ascending=False).head(10)
print(tabulate(top_reviews, headers='keys', tablefmt='psql', showindex=False))
```

运行这段代码，结果如下所示：

name	pageRank	stars
Chris	5.84	3
Lorrie	4.95	2
Dani	3.47	1
Victor	3.35	3
Francine	2.93	3
Rex	2.79	2
Jon	2.55	3
Rachel	2.47	3
Leslie	2.46	2
Benay	2.46	3

排名最高的用户 Chris 和 Lorrie 给了 Bellagio 酒店较低的评级，他们位列前 1000 名最具影响力的用户（根据之前的查询结果），因此也许有必要单独接洽。此外，由于许多评论者会在其逗留期间撰写评论，因此实时向商家提醒有影响力的评论者，可能会促进双方互动。

Bellagio 交叉推广

在找到有影响力的评论者之后，Bellagio 酒店现在要求我们通过那些人脉广的客户帮助其确定其他交叉推广业务。我们的方案是建议他们吸引不同社团中的新客户以巩固客户基础，这是新的机会。可以用之前介绍过的中间中心性算法计算，找出有哪些评论过 Bellagio 酒店的人不仅在整个 Yelp 网络中联系广泛，还可以作为不同群组之间的"桥梁"。

我们只想在拉斯维加斯查找有影响力的人，所以首先标记这些用户：

```
MATCH (u:User)
WHERE exists((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CITY]->
                              (:City {name: "Las Vegas"}))
SET u:LasVegas
```

对拉斯维加斯的用户运行中间中心性算法比较耗时，因此我们将使用其变体，即 RA-Brandes 算法。该算法根据采样节点计算出中间中心性得分，而且对最短路径的搜索只到一定深度。

通过几次实验，我们用与默认值不同的参数设置改进结果。我们使用最多 4 跳（maxDepth 为 4）的最短路径，并且抽样 20% 的节点（probability 为 0.2）。注意，增加跳数和节点数量通常会提高准确率，但是计算更耗时。针对特定问题，通常需要测试最优参数以确定收益递减点。

下面的查询将运行该算法，并将结果存储在 between 性质中：

```
CALL algo.betweenness.sampled('LasVegas', 'FRIENDS',
  {write: true, writeProperty: "between", maxDepth: 4, probability: 0.2}
)
```

在用这些分值进行查询之前，可先写一个探查查询，了解这些分值的分布状况：

```
MATCH (u:User)
WHERE exists(u.between)
RETURN count(u.between) AS count,
       avg(u.between) AS ave,
       toInteger(percentileDisc(u.between, 0.5)) AS `50%`,
       toInteger(percentileDisc(u.between, 0.75)) AS `75%`,
       toInteger(percentileDisc(u.between, 0.90)) AS `90%`,
       toInteger(percentileDisc(u.between, 0.95)) AS `95%`,
       toInteger(percentileDisc(u.between, 0.99)) AS `99%`,
       toInteger(percentileDisc(u.between, 0.999)) AS `99.9%`,
       toInteger(percentileDisc(u.between, 0.9999)) AS `99.99%`,
       toInteger(percentileDisc(u.between, 0.99999)) AS `99.999%`,
       toInteger(percentileDisc(u.between, 1)) AS p100
```

执行上述查询，输出结果如下所示：

count	ave	50%	75%	90%	95%	99%	99.9%	99.99%	99.999%	100%
506028	320538.6014	0	10005	318944	1001655	4436409	34854988	214080923	621434012	1998032952

有一半用户的得分为 0，这意味着他们的人脉并不是很广；而前 1%（99% 列）的用户都位于 50 万用户之间的 400 多万条最短路径上。综合考虑，大多数用户的关系网很差，但有少数用户对信息具有强有力的控制，这正是小世界网络的典型表现。

执行以下查询，找出谁是“超级连接者”：

```
MATCH(u:User)-[:WROTE]->()-[:REVIEWS]->(:Business {name:"Bellagio Hotel"})
WHERE exists(u.between)
RETURN u.name AS user,
       toInteger(u.between) AS betweenness,
       u.hotelPageRank AS pageRank,
       size((u)-[:WROTE]->()-[:REVIEWS]->()-[:IN_CATEGORY]->
                              (:Category {name: "Hotels"}))
       AS hotelReviews
ORDER BY u.between DESC
LIMIT 10
```

输出结果如下所示：

user	betweenness	pageRank	hotelReviews
Misti	841707563	12.239516000000004	19
Christine	236269693	10.740770499999998	16
Erica	235806844	10.993773	6
Mike	215534452	NULL	2
J	192155233	11.431505999999997	103
Michael	161335816	5.105143	31
Jeremy	160312436	9.576763499999998	6
Michael	139960910	11.460049	13
Chris	136697785	5.838922499999999	5
Connie	133372418	9.118103499999998	7

可以看到一些与前述 PageRank 分值查询相同的人，Mike 是有趣的个例。本次计算将其排除在外，这是因为他没有评价过足够多的（至少 3 家）酒店，但他似乎在拉斯维加斯的 Yelp 用户圈里人脉很广。

为了触达更多用户，还要研究这些“连接者”所表现出的其他偏好，了解应该推广的内容。其中很多用户也评价过餐馆，所以可用以下查询来查找他们最喜欢的餐馆：

```
// 查找评论过Bellagio酒店的前50位用户
MATCH (u:User)-[:WROTE]->()-[:REVIEWS]->(:Business {name:"Bellagio Hotel"})
WHERE u.between > 4436409
WITH u ORDER BY u.between DESC LIMIT 50

// 查找有用户评论过的拉斯维加斯餐馆
MATCH (u)-[:WROTE]->(review)-[:REVIEWS]-(business)
WHERE (business)-[:IN_CATEGORY]->(:Category {name: "Restaurants"})
AND   (business)-[:IN_CITY]->(:City {name: "Las Vegas"})

// 仅查找用户评论数大于3的餐馆
WITH business, avg(review.stars) AS averageReview, count(*) AS numberOfReviews
WHERE numberOfReviews >= 3
```

```
RETURN business.name AS business, averageReview, numberOfReviews
ORDER BY averageReview DESC, numberOfReviews DESC
LIMIT 10
```

该查询查找了前 50 名有影响力的“连接者”，并且查找排名前 10 且至少被其中 3 人评级过的拉斯维加斯餐馆。执行该查询，输出结果如下所示：

business	averageReview	numberOfReviews
Jean Georges Steakhouse	5.0	6
Sushi House Goyemon	5.0	6
Art of Flavors	5.0	4
é by José Andrés	5.0	4
Parma By Chef Marc	5.0	4
Yonaka Modern Japanese	5.0	4
Kabuto	5.0	4
Harvest by Roy Ellamar	5.0	3
Portofino by Chef Michael LaPlaca	5.0	3
Montesano’s Eateria	5.0	3

基于此，可以建议 Bellagio 酒店与这些餐馆联合推出促销活动，以吸引那些通常无法触达的群体中的新顾客。给 Bellagio 酒店评级的“超级连接者”是很好的代理人，有助于评估哪些餐馆可能会吸引新类型的目标顾客。

在帮助 Bellagio 酒店触达新的顾客群组后，接下来研究如何使用社团发现算法来进一步改进应用程序。

7.1.7 查找相似类别

当终端用户使用这款应用程序查找酒店时，我们希望向其展示他们可能感兴趣的其他商家。Yelp 数据集包含 1000 多个类别，其中一些类别很相似。可以利用这种相似性在应用程序中向用户推荐他们可能感兴趣的新商家。

在当前的图模型中，类别之间没有任何关系，但是可以运用 2.3.6 节所介绍的思想，基于商家自己确定的分类来构建一个类别相似图。

假设一个商家同时属于 `Hotels` 类和 `Historical Tours` 类，如图 7-8 所示。

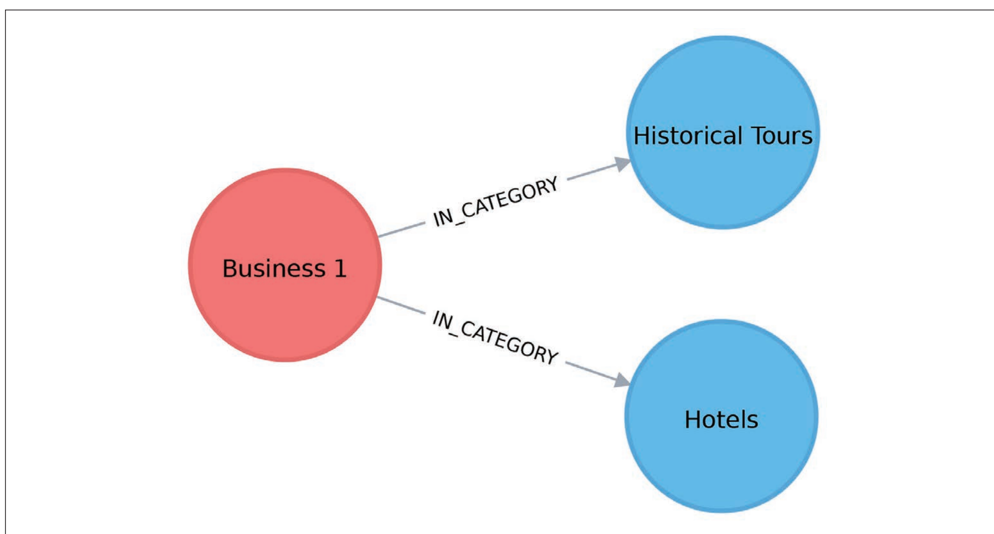


图 7-8：涉及两个类别的商家

这可以产生一个投影图，其中 `Hotels` 和 `Historical Tours` 这两个类别之间有一条权重为 1 的关联关系，如图 7-9 所示。

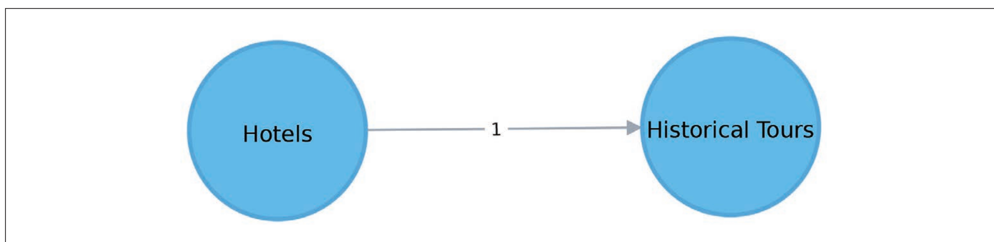


图 7-9：类别投影图

在这种情况下，实际上并不需要创建类别相似图；相反，可以在投影相似图上运行标签传播算法等社团发现算法。使用标签传播算法可以有效地将商家聚类到与其最相近的超类别：

```
CALL algo.labelPropagation.stream(
  'MATCH (c:Category) RETURN id(c) AS id',
  'MATCH (c1:Category)<-[:IN_CATEGORY]-()-[:IN_CATEGORY]->(c2:Category)
   WHERE id(c1) < id(c2)
   RETURN id(c1) AS source, id(c2) AS target, count(*) AS weight',
  {graph: "cypher"}
)
YIELD nodeId, label
MATCH (c:Category) WHERE id(c) = nodeId
MERGE (sc:SuperCategory {name: "SuperCategory-" + label})
MERGE (c)-[:IN_SUPER_CATEGORY]->(sc)
```

可以为这些超类别起一个更友好的名称，以便涵盖大多数类别：

```
MATCH (sc:SuperCategory)<-[:IN_SUPER_CATEGORY]-(category)
WITH sc, category, size((category)<-[:IN_CATEGORY]-()) as size
ORDER BY size DESC
WITH sc, collect(category.name)[0] as biggestCategory
SET sc.friendlyName = "SuperCat " + biggestCategory
```

类别和超类别如图 7-10 所示。

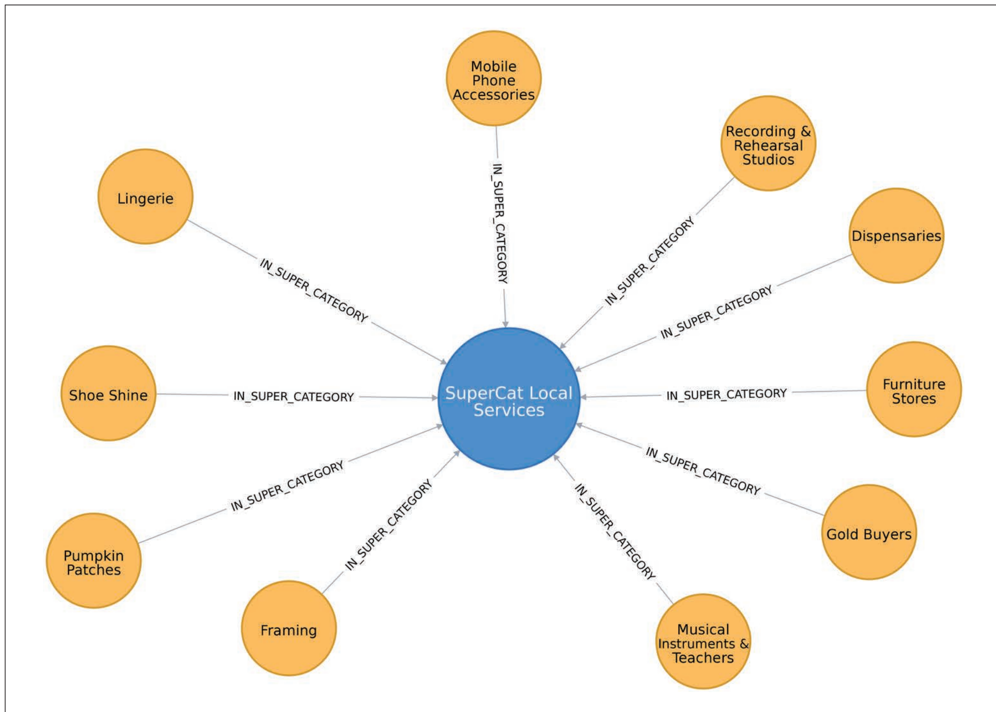


图 7-10：类别和超类别

下面的查询可以查找拉斯维加斯当地与 Hotels 最相似的类别：

```
MATCH (hotels:Category {name: "Hotels"}),
      (lasVegas:City {name: "Las Vegas"}),
      (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-
                                   (otherCategory)
RETURN otherCategory.name AS otherCategory,
       size((otherCategory)<-[:IN_CATEGORY]-(:Business)-
                               [:IN_CITY]->(lasVegas)) AS businesses
ORDER BY count DESC
LIMIT 10
```

执行该查询，输出如下所示：

otherCategory	businesses
Tours	189
Car Rental	160
Limos	84
Resorts	73
Airport Shuttles	52
Taxis	35
Vacation Rentals	29
Airports	25
Airlines	23
Motorcycle Rental	19

这些结果看起来奇怪吗？旅游（Tours）和出租车（Taxis）显然都不是酒店，但请记住，这是基于自行申报的分类得到的结果。标签传播算法在该相似分组中真正揭示的是邻近的商家和服务。

下面看看在这些类别中哪些商家的评级高于平均水平：

```
// 查找与酒店同属某个超类别的拉斯维加斯商家
MATCH (hotels:Category {name: "Hotels"}),
      (hotels)-[:IN_SUPER_CATEGORY]->()<-[:IN_SUPER_CATEGORY]-
                                   (otherCategory),
      (otherCategory)<-[:IN_CATEGORY]-(business)
WHERE (business)-[:IN_CITY]->(:City {name: "Las Vegas"})

// 随机选择10个类别并且计算第90百分位星级评级
WITH otherCategory, count(*) AS count,
     collect(business) AS businesses,
     percentileDisc(business.averageStars, 0.9) AS p90Stars
ORDER BY rand() DESC
LIMIT 10

// 使用模式理解从平均评级高于第90百分位的那些类别中选择商家
// 每个类别选择一个商家
WITH otherCategory, [b in businesses where b.averageStars >= p90Stars]
                    AS businesses

// Select one business per category
WITH otherCategory, businesses[toInteger(rand() * size(businesses))] AS business

RETURN otherCategory.name AS otherCategory,
       business.name AS business,
       business.averageStars AS averageStars
```

在该查询中首次用到了**模式理解**。模式理解是一种基于模式匹配创建列表的语法结构。它使用 MATCH 子句和 WHERE 子句为谓词查找特定模式，然后生成一个自定义投影。Cypher 的这一功能是受 GraphQL（一种 API 查询语言）的启发而增加的。

执行该查询，结果如下所示：

otherCategory	business	averageStars
Motorcycle Rental	Adrenaline Rush Slingshot Rentals	5.0
Snorkeling	Sin City Scuba	5.0
Guest Houses	Hotel Del Kacvinsky	5.0
Car Rental	The Lead Team	5.0
Food Tours	Taste BUZZ Food Tours	5.0
Airports	Signature Flight Support	5.0
Public Transportation	JetSuiteX	4.6875
Ski Resorts	Trikke Las Vegas	4.833333333333332
Town Car Service	MW Travel Vegas	4.866666666666665
Campgrounds	McWilliams Campground	3.875

然后可以根据用户在应用程序中的即时行为进行实时推荐，例如当用户关注拉斯维加斯的酒店时，可以突出显示该城市各类与之邻近的好评商家。还可以将这些方法推广到任何地方的任何业务类别（例如餐馆或剧院）。

练　　习

- 你能描绘出对某个城市的酒店的评论随着时间如何变化吗?
- 对于某家特定的酒店或其他商家，又如何呢?
- 人气变化体现了什么趋势（季节性趋势或其他趋势）?
- 最具影响力的评论者是否仅连接（出连接）到其他有影响力的评论者?

7.2　使用Spark分析航班数据

下面利用另一个场景演示如何使用 Spark 分析美国机场数据。假设你是数据科学家，拥有大量行程计划表，希望深入挖掘航空公司的航班数据和航班延误信息。首先研究机场信息和航班信息，然后深入研究两个特定机场的航班延误情况。我们将使用社团发现算法分析航线并且探寻飞行常客积分的最佳使用方法。

可从美国交通部获取大量交通信息。我们将使用 2018 年 5 月的航旅准时性数据进行分析，其中涉及当月从美国出发和到达美国的航班。为了增加关于机场的更多详细信息（比如位置信息），还将从独立数据源 OpenFlights 加载数据。

先把数据加载到 Spark 中。如前所示，这些数据保存在 CSV 文件中，详见本书配套文件。

```
nodes = spark.read.csv("data/airports.csv", header=False)

cleaned_nodes = (nodes.select("_c1", "_c3", "_c4", "_c6", "_c7")
                 .filter("_c3 = 'United States'")
                 .withColumnRenamed("_c1", "name")
                 .withColumnRenamed("_c4", "id")s
                 .withColumnRenamed("_c6", "latitude")
                 .withColumnRenamed("_c7", "longitude")
                 .drop("_c3"))
cleaned_nodes = cleaned_nodes[cleaned_nodes["id"] != "\\N"]

relationships = spark.read.csv("data/188591317_T_ONTIME.csv", header=True)

cleaned_relationships = (relationships
                         .select("ORIGIN", "DEST", "FL_DATE", "DEP_DELAY",
                                 "ARR_DELAY", "DISTANCE", "TAIL_NUM", "FL_NUM",
                                  "CRS_DEP_TIME", "CRS_ARR_TIME",
                                  "UNIQUE_CARRIER")
                         .withColumnRenamed("ORIGIN", "src")
                         .withColumnRenamed("DEST", "dst")
                         .withColumnRenamed("DEP_DELAY", "deptDelay")
                         .withColumnRenamed("ARR_DELAY", "arrDelay")
                         .withColumnRenamed("TAIL_NUM", "tailNumber")
                         .withColumnRenamed("FL_NUM", "flightNumber")
                         .withColumnRenamed("FL_DATE", "date")
                         .withColumnRenamed("CRS_DEP_TIME", "time")
                         .withColumnRenamed("CRS_ARR_TIME", "arrivalTime")
                         .withColumnRenamed("DISTANCE", "distance")
                         .withColumnRenamed("UNIQUE_CARRIER", "airline")
                         .withColumn("deptDelay",
                              F.col("deptDelay").cast(FloatType()))
                         .withColumn("arrDelay",
                              F.col("arrDelay").cast(FloatType()))
                         .withColumn("time", F.col("time").cast(IntegerType()))
                         .withColumn("arrivalTime",
                              F.col("arrivalTime").cast(IntegerType()))
                         )

g = GraphFrame(cleaned_nodes, cleaned_relationships)
```

因为一些机场的机场代码不适用，所以必须对节点做一些整理工作。要采用更便于描述的列名，还要将某些项转换为恰当的数值类型。按照 Spark 的 GraphFrames 库的要求，还需要确保有名为 `id`、`dst` 和 `src` 的列。

我们还将单独创建一个 DataFrame 对象，将航空公司代码映射到航空公司名称，稍后会用到该对象。

```
airlines_reference = (spark.read.csv("data/airlines.csv")
      .select("_c1", "_c3")
      .withColumnRenamed("_c1", "name")
      .withColumnRenamed("_c3", "code"))

airlines_reference = airlines_reference[airlines_reference["code"] != "null"]
```

7.2.1　探索性分析

首先用探索性分析了解数据概况，比如机场数量：

```
g.vertices.count()
```

```
1435
```

这些机场之间有多少连接？

```
g.edges.count()
```

```
616529
```

7.2.2　热门机场

哪些机场的出港航班最多？可以使用度中心性算法计算出港航班的数量：

```
airports_degree = g.outDegrees.withColumnRenamed("id", "oId")

full_airports_degree = (airports_degree
                        .join(g.vertices, airports_degree.oId == g.vertices.id)
                        .sort("outDegree", ascending=False)
                        .select("id", "name", "outDegree"))

full_airports_degree.show(n=10, truncate=False)
```

运行以上代码，结果如下所示：

id	name	outDegree
ATL	Hartsfield Jackson Atlanta International Airport	33837
ORD	Chicago O'Hare International Airport	28338
DFW	Dallas Fort Worth International Airport	23765
CLT	Charlotte Douglas International Airport	20251
DEN	Denver International Airport	19836
LAX	Los Angeles International Airport	19059
PHX	Phoenix Sky Harbor International Airport	15103
SFO	San Francisco International Airport	14934
LGA	La Guardia Airport	14709
IAH	George Bush Intercontinental Houston Airport	14407

其中的机场涉及美国的多个大都市，如芝加哥、亚特兰大、洛杉矶和纽约，它们都有热门机场。还可以使用以下代码把出港航班可视化。

```
plt.style.use('fivethirtyeight')

ax = (full_airports_degree
```

```
            .toPandas()
            .head(10)
            .plot(kind='bar', x='id', y='outDegree', legend=None))

ax.xaxis.set_label_text("")
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

可视化结果如图 7-11 所示。

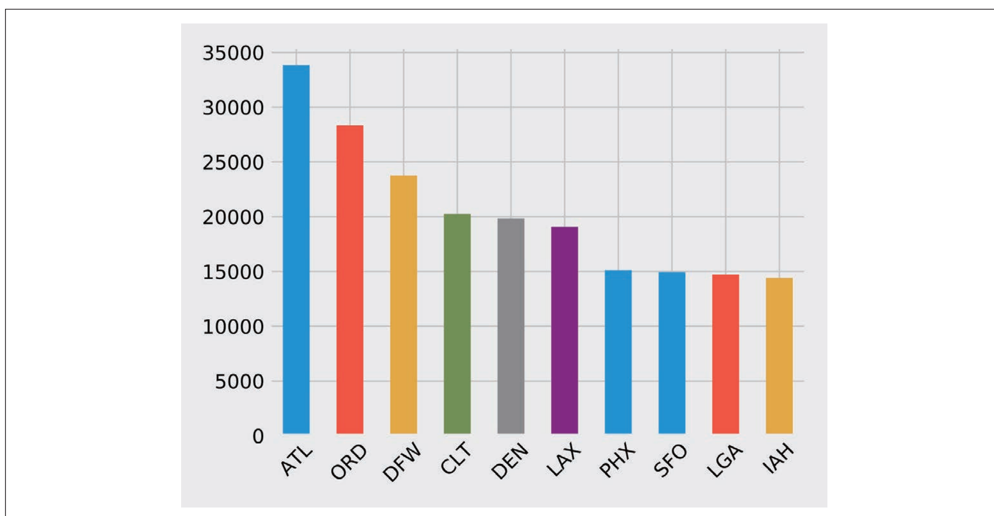


图 7-11：机场出港航班

航班量的差异很明显。排名第 5 的丹佛国际机场（DEN）的出港航班数量仅约为排名第 1 的哈兹菲尔德 – 杰克逊亚特兰大国际机场（ATL）的一半。

7.2.3 源自ORD的延误

假设我们经常要往返于美国东西海岸之间，所以希望了解经过像芝加哥奥黑尔国际机场（ORD）这样的中间枢纽所导致的延误。由于该数据集也包含了航班延误数据，因此可以直接开展研究。

以下代码查找 ORD 出港航班的平均延误时间，并按目的地机场进行分组：

```
delayed_flights = (g.edges
                   .filter("src = 'ORD' and deptDelay > 0")
                   .groupBy("dst")
                   .agg(F.avg("deptDelay"), F.count("deptDelay"))
                   .withColumn("averageDelay",
                               F.round(F.col("avg(deptDelay)"), 2))
                   .withColumn("numberOfDelays",
```

```
                                    F.col("count(deptDelay)")))

(delayed_flights
 .join(g.vertices, delayed_flights.dst == g.vertices.id)
 .sort(F.desc("averageDelay"))
 .select("dst", "name", "averageDelay", "numberOfDelays")
 .show(n=10, truncate=False))
```

在按目的地机场分组计算平均延迟后，可将 Spark 生成的 DataFrame 对象与包含所有机场（顶点）的 DataFrame 进行连接操作，打印出目的地机场的全名。

运行以上代码将返回航班延误最严重的 10 个目的地机场，如下所示：

dst	name	averageDelay	numberOfDelays
CKB	North Central West Virginia Airport	145.08	12
OGG	Kahului Airport	119.67	9
MQT	Sawyer International Airport	114.75	12
MOB	Mobile Regional Airport	102.2	10
TTN	Trenton Mercer Airport	101.18	17
AVL	Asheville Regional Airport	98.5	28
ISP	Long Island Mac Arthur Airport	94.08	13
ANC	Ted Stevens Anchorage International Airport	83.74	23
BTV	Burlington International Airport	83.2	25
CMX	Houghton County Memorial Airport	79.18	17

这很有趣，有一项数据确实很突出：从 ORD 到 CKB 的 12 次航班平均延误超过两小时！下面查找这两个机场之间的航班，看看发生了什么：

```
from_expr = 'id = "ORD"'
to_expr = 'id = "CKB"'
ord_to_ckb = g.bfs(from_expr, to_expr)

ord_to_ckb = ord_to_ckb.select(
  F.col("e0.date"),
  F.col("e0.time"),
  F.col("e0.flightNumber"),
  F.col("e0.deptDelay"))
```

可以用以下代码绘制航班信息：

```
ax = (ord_to_ckb
      .sort("date")
      .toPandas()
      .plot(kind='bar', x='date', y='deptDelay', legend=None))

ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

运行这段代码，得到图 7-12 所示的图表。

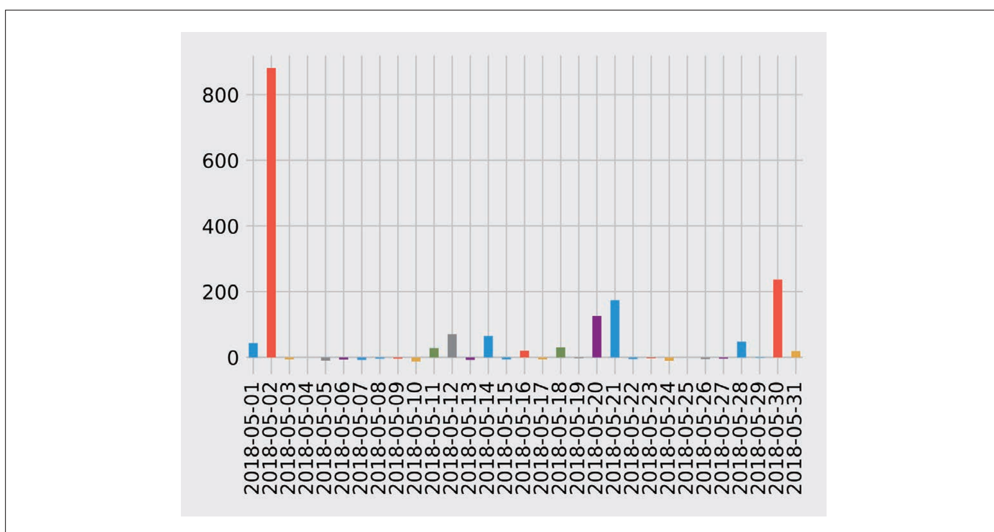


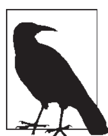
图 7-12：从 ORD 到 CKB 的航班

大约有一半航班延误，但 2018 年 5 月 2 日的延误超过 14 小时，这大大地扭曲了平均值。

如果想查找某个沿海机场进出港航班的延误信息要怎么办？这些机场经常受到恶劣天气条件的影响，所以可能会发生一些延误。

7.2.4 SFO的糟糕一天

来看旧金山国际机场（SFO）的航班延误情况，该机场因大雾导致的低能见度问题而闻名。分析方法之一就是研究**模体**（motif），即重复出现的子图或模式。



Neo4j 中的图模式与模体等价，可通过 `MATCH` 子句或 Cypher 中的模式表达式发现。

GraphFrames 支持对模体的搜索，因此可以将航班结构用作查询的一部分。下面用模体来查找 2018 年 5 月 11 日进出 SFO 延误最严重的航班，代码如下：

```
motifs = (g.find("(a)-[ab]->(b); (b)-[bc]->(c)")
          .filter("""(b.id = 'SFO') and
                  (ab.date = '2018-05-11' and bc.date = '2018-05-11') and
                  (ab.arrDelay > 30 or bc.deptDelay > 30) and
                  (ab.flightNumber = bc.flightNumber) and
                  (ab.airline = bc.airline) and
                  (ab.time < bc.time)"""))
```

模体 (a)-[ab]->(b); (b)-[bc]->(c) 用于查找进出同一机场的航班。然后，筛选所生成的模式以查找航班，要求如下：

- 有先后顺序，第 1 架航班到达 SFO 的时间要早于第 2 架航班离开 SFO 的时间；
- 到达或离开 SFO 的延误时间超过 30 分钟；
- 航班号和航空公司相同。

得到结果并选择感兴趣的列：

```
result = (motifs.withColumn("delta", motifs.bc.deptDelay - motifs.ab.arrDelay)
          .select("ab", "bc", "delta")
          .sort("delta", ascending=False))

result.select(
    F.col("ab.src").alias("a1"),
    F.col("ab.time").alias("a1DeptTime"),
    F.col("ab.arrDelay"),
    F.col("ab.dst").alias("a2"),
    F.col("bc.time").alias("a2DeptTime"),
    F.col("bc.deptDelay"),
    F.col("bc.dst").alias("a3"),
    F.col("ab.airline"),
    F.col("ab.flightNumber"),
    F.col("delta")
).show()
```

我们还计算了航班进港和出港之间的**时间差**（delta），以确定哪些延误可以真正归因于 SFO。

运行代码，结果如下所示：

airline	flightNumber	a1	a1DeptTime	arrDelay	a2	a2DeptTime	deptDelay	a3	delta
WN	1454	PDX	1130	-18.0	SFO	1350	178.0	BUR	196.0
OO	5700	ACV	1755	-9.0	SFO	2235	64.0	RDM	73.0
UA	753	BWI	700	-3.0	SFO	1125	49.0	IAD	52.0
UA	1900	ATL	740	40.0	SFO	1110	77.0	SAN	37.0
WN	157	BUR	1405	25.0	SFO	1600	39.0	PDX	14.0
DL	745	DTW	835	34.0	SFO	1135	44.0	DTW	10.0
WN	1783	DEN	1830	25.0	SFO	2045	33.0	BUR	8.0
WN	5789	PDX	1855	119.0	SFO	2120	117.0	DEN	-2.0
WN	1585	BUR	2025	31.0	SFO	2230	11.0	PHX	-20.0

出现在第 1 行的是延误情况最严重的 WN1454，它很早到达，但出港时间晚了近 3 小时。还可以看到，在 `arrDelay` 列中有一些负值，这意味着航班提前到达了 SFO。

还要注意，有些航班（如 WN5789 和 WN1585）要在 SFO 补时，其 `delta` 值为负。

7.2.5　通过航空公司互连的机场

现在假设我们已经旅行过很多次，并且想用飞行常客积分尽可能高效地游览更多目的地，因为这些积分很快就要到期了。如果从美国某个机场出发，需要经过多少个机场，才可以乘坐同一家航空公司的航班返回出发机场呢？

先找出所有航空公司，计算出每家航空公司的航班数量：

```
airlines = (g.edges
 .groupBy("airline")
 .agg(F.count("airline").alias("flights"))
 .sort("flights", ascending=False))

full_name_airlines = (airlines_reference
                      .join(airlines, airlines.airline
                            == airlines_reference.code)
                      .select("code", "name", "flights"))
```

创建一张条形图来展示航空公司的情况：

```
ax = (full_name_airlines.toPandas()
      .plot(kind='bar', x='name', y='flights', legend=None))

ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

执行该查询，输出结果如图 7-13 所示。

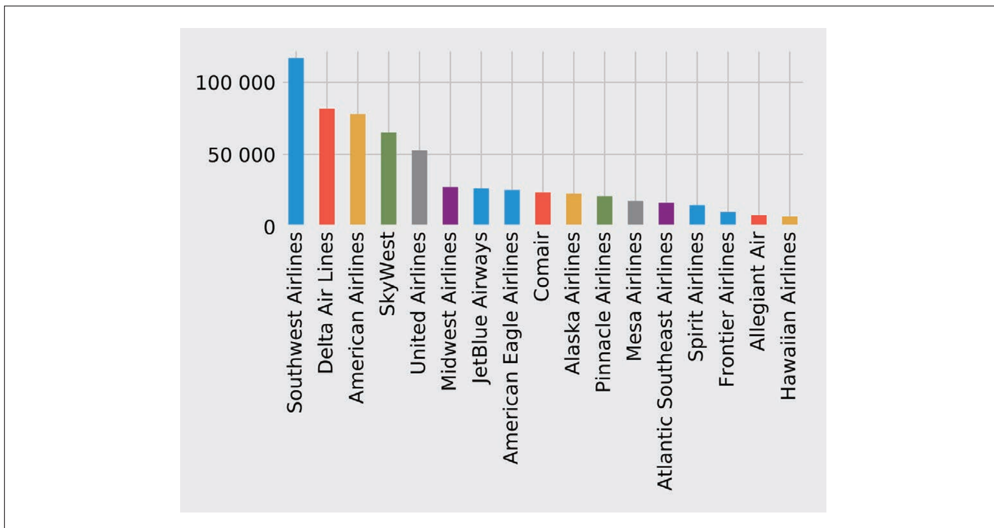


图 7-13：航空公司的航班数量

下面编写一个函数，用强连通分量算法查找每家航空公司的机场分组，且分组中所有机场都有往返该分组中其他机场的航班：

```
def find_scc_components(g, airline):
    # 创建只包含给定航空公司航班的子图
    airline_relationships = g.edges[g.edges.airline == airline]
    airline_graph = GraphFrame(g.vertices, airline_relationships)

    # Calculate the Strongly Connected Components
    scc = airline_graph.stronglyConnectedComponents(maxIter=10)

    # 查找最大分量的大小并返回
    return (scc
        .groupBy("component")
        .agg(F.count("id").alias("size"))
        .sort("size", ascending=False)
        .take(1)[0]["size"])
```

编写以下代码创建一个 DataFrame 对象，其中包含每家航空公司及其最大强连通分量中的机场数量：

```
# 计算每家航空公司的最大强连通分量
airline_scc = [(airline, find_scc_components(g, airline))
               for airline in airlines.toPandas()["airline"].tolist()]
airline_scc_df = spark.createDataFrame(airline_scc, ['id', 'sccCount'])

# 对DataFrame对象SCC和airlines进行连接运算，获取航空公司的航班数量
# 及其最大的分量中可达机场的数量
airline_reach = (airline_scc_df
 .join(full_name_airlines, full_name_airlines.code == airline_scc_df.id)
 .select("code", "name", "flights", "sccCount")
 .sort("sccCount", ascending=False))
```

然后创建条形图来展示航空公司信息：

```
ax = (airline_reach.toPandas()
      .plot(kind='bar', x='name', y='sccCount', legend=None))

ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

执行该查询，输出如图 7-14 所示。

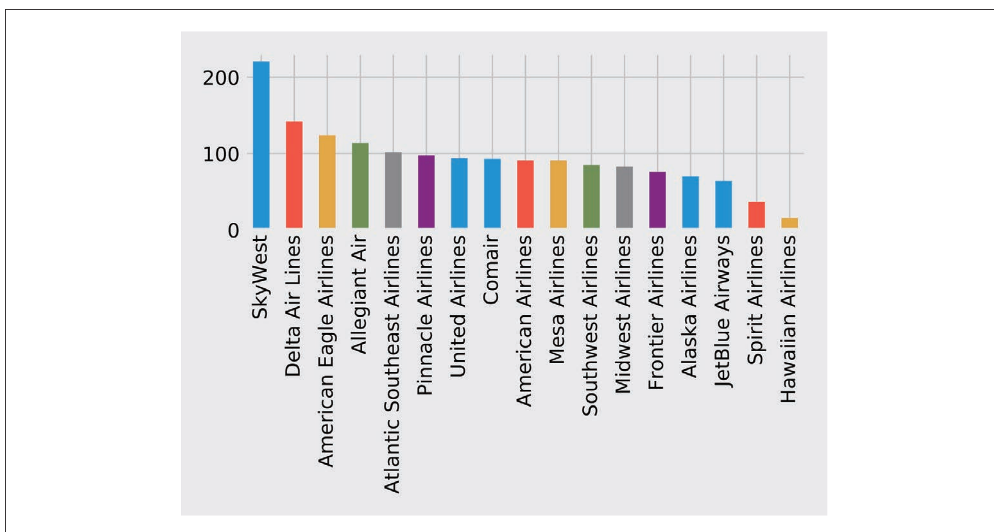


图 7-14：航空公司可达机场的数量

一方面，SkyWest 公司拥有最大的社团，其中有 200 多个强连通的机场。这在一定程度上反映出它作为一家联营航空公司的商业模式：为合作的航空公司运营飞机。另一方面，Southwest 公司的航班数量虽然最多，但是仅连通了大约 80 个机场。

假设我们拥有的大部分飞行常客积分来自达美航空公司（DL），能在网络中为特定航空公司找到那些形成社团的机场吗？

```
airline_relationships = g.edges.filter("airline = 'DL'")
airline_graph = GraphFrame(g.vertices, airline_relationships)

clusters = airline_graph.labelPropagation(maxIter=10)
(clusters
 .sort("label")
 .groupby("label")
 .agg(F.collect_list("id").alias("airports"),
      F.count("id").alias("count"))
 .sort("count", ascending=False)
 .show(truncate=70, n=10))
```

执行该查询，结果如下所示：

label	airports	count
1606317768706	[IND, ORF, ATW, RIC, TRI, XNA, ECP, AVL, JAX, SYR, BHM, GSO, MEM, C...	89
1219770712067	[GEG, SLC, DTW, LAS, SEA, BOS, MSN, SNA, JFK, TVC, LIH, JAC, FLL, M...	53
17179869187	[RHV]	1
25769803777	[CWT]	1
25769803776	[CDW]	1

25769803782	[KNW]	1
25769803778	[DRT]	1
25769803779	[FOK]	1
25769803781	[HVR]	1
42949672962	[GTF]	1

DL 使用的大多数机场可聚类为两个分组，下面深入研究。因为要展示的机场太多，所以只展示度（进出港航班数）最大的机场。编写以下代码来计算机场的度：

```
all_flights = g.degrees.withColumnRenamed("id", "aId")
```

然后将其与属于最大簇的机场合并处理：

```
(clusters
 .filter("label=1606317768706")
 .join(all_flights, all_flights.aId == result.id)
 .sort("degree", ascending=False)
 .select("id", "name", "degree")
 .show(truncate=False))
```

执行该查询，输出结果如下所示：

id	name	degree
DFW	Dallas Fort Worth International Airport	47514
CLT	Charlotte Douglas International Airport	40495
IAH	George Bush Intercontinental Houston Airport	28814
EWR	Newark Liberty International Airport	25131
PHL	Philadelphia International Airport	20804
BWI	Baltimore/Washington International Thurgood Marshall Airport	18989
MDW	Chicago Midway International Airport	15178
BNA	Nashville International Airport	12455
DAL	Dallas Love Field	12084
IAD	Washington Dulles International Airport	11566
STL	Lambert St Louis International Airport	11439
HOU	William P Hobby Airport	9742
IND	Indianapolis International Airport	8543
PIT	Pittsburgh International Airport	8410
CLE	Cleveland Hopkins International Airport	8238
CMH	Port Columbus International Airport	7640
SAT	San Antonio International Airport	6532
JAX	Jacksonville International Airport	5495
BDL	Bradley International Airport	4866
RSW	Southwest Florida International Airport	4569

现在对第 2 大簇执行相同的操作：

```
(clusters
 .filter("label=1219770712067")
 .join(all_flights, all_flights.aId == result.id)
 .sort("degree", ascending=False)
 .select("id", "name", "degree")
 .show(truncate=False))
```

执行该查询，结果如下所示：

id	name	degree
ATL	Hartsfield Jackson Atlanta International Airport	67672
ORD	Chicago O'Hare International Airport	56681
DEN	Denver International Airport	39671
LAX	Los Angeles International Airport	38116
PHX	Phoenix Sky Harbor International Airport	30206
SFO	San Francisco International Airport	29865
LGA	La Guardia Airport	29416
LAS	McCarran International Airport	27801
DTW	Detroit Metropolitan Wayne County Airport	27477
MSP	Minneapolis-St Paul International/Wold-Chamberlain Airport	27163
BOS	General Edward Lawrence Logan International Airport	26214
SEA	Seattle Tacoma International Airport	24098
MCO	Orlando International Airport	23442
JFK	John F Kennedy International Airport	22294
DCA	Ronald Reagan Washington National Airport	22244
SLC	Salt Lake City International Airport	18661
FLL	Fort Lauderdale Hollywood International Airport	16364
SAN	San Diego International Airport	15401
MIA	Miami International Airport	14869
TPA	Tampa International Airport	12509

当查看 DL 网站上的飞行常客计划时，发现有一个“用二送一”的促销活动。如果在两次飞行中都使用积分，就可以免费再飞一次——但是只能在上述两个簇之一的航线上飞行。也许仅在一个簇的航线上飞行可以更好地利用时间，当然还有积分。

练　习

- 使用最短路径算法计算从你所在的城市到博兹曼黄石国际机场（BZN）的航班数量。
- 如果使用关系权重，会有什么不同吗？

7.3 小结

前几章详细介绍了如何在 Spark 和 Neo4j 中运用路径查找算法、中心性算法和社团发现算法等重要的图算法。本章介绍了在实际的任务和分析中综合运用这几种算法的工作流程。我们利用旅游业务场景说明了如何在 Neo4j 中分析 Yelp 数据，还通过个人航旅场景学习了如何在 Spark 中分析美国航空公司数据。

接下来将探讨图增强机器学习，它已经逐渐成为图算法的一项重要应用。

第 8 章
使用图算法增强机器学习

前面介绍了几种在每次迭代中学习和更新状态的算法，比如标签传播算法。然而到目前为止，重点一直放在用于常规分析的图算法上。鉴于图在机器学习中的应用越来越多，下面研究如何使用图算法来改进机器学习工作流程。

本章将重点介绍使用图算法改进机器学习预测的实用方法：关联特征提取及其在预测关系中的应用。首先介绍机器学习的一些基本概念，阐述上下文数据对于预测的重要性；然后简单介绍图特征的应用方式，包括在垃圾邮件检测、欺诈检测和链接预测中的应用。

本章将介绍如何创建一个机器学习管道，然后训练并评估链接预测模型，并将 Neo4j 和 Spark 集成到工作流程中。本章示例基于引文网络数据集，其中包含作者、论文、作者关系和引用关系。我们将使用几个模型来预测论文作者之间将来有无可能合作，并演示如何用图算法改进结果。

8.1　机器学习和上下文的重要性

机器学习不是人工智能，而是实现人工智能的方法。机器学习使用算法通过特定实例和基于预期结果的逐步改进来训练软件，不需要显式编程就能得到不错的结果。在训练时需要向模型提供大量数据，并且要让机器能够学会如何处理和整合这些信息。

从这个意义上讲，学习意味着算法的迭代，通过不断改变来接近某一客观目标，比如与训练数据相比减少分类错误。机器学习也是动态的，当给予更多数据时能够修改和优化自身。在使用前可进行多批次训练，而在使用过程中可进行在线学习。

最近在机器学习预测、大规模数据集的可访问性以及并行计算能力等方面的突出表现，使得机器学习对于为人工智能应用开发概率模型来说变得更加实用。随着机器学习的普及，应牢记其基本目标：像人类一样做出选择。如果忘记了该目标，可能只会得到另一个高度定向、基于规则的软件版本。

为了提高机器学习的准确率，并使解决方案得到广泛应用，需要整合大量上下文信息，就像人们可以使用上下文更好地做出决策一样。人们通常会利用周围的上下文信息，而并不仅限于直接的数据，借此找出某种情况下的关键信息，估计缺失信息，并决定如何将经验应用于新情况。上下文有助于改进预测结果。

图、上下文和准确率

在没有外围信息和关联信息的情况下，解决方案要试图预测行为或针对不同情况提出建议，必须经过充分训练并具有合理的规则。这在某种程度上说明了为何人工智能擅长具体、定义良好的任务，而难以处理二义性问题。图增强机器学习有助于填补缺少的上下文信息，这些信息对于更好地决策非常重要。

在图论和现实生活中，关系往往是预测行为的重要因素。例如一个人投了票，那么他的朋友、家人甚至同事等投票的可能性就会增加。图 8-1 展示了一种基于投票情况和 Facebook 好友的连锁反应，来自 Robert Bond 等人于 2012 年发表的一篇论文。

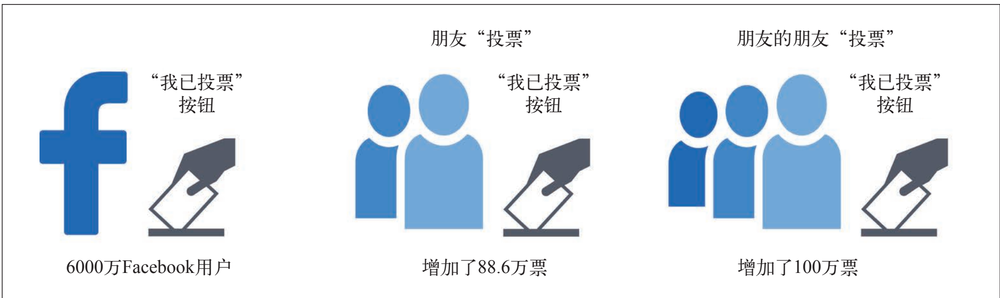


图 8-1：人们的投票受社交网络影响。在本例中，两跳之外的朋友比直接关系的影响更大

论文作者发现，公开了投票情况的朋友影响了 1.4% 宣称投过票的用户，有趣的是，朋友的朋友的影响为 1.7%。小百分比也可以产生重要影响，图 8-1 显示，两跳之外的那些人总的影响比直接朋友更大。Nicholas Christakis 和 James Fowler 合著的《大连接》一书谈到了投票问题以及其他一些受社交网络影响的示例。

添加图特征和上下文可以改进预测结果，特别是在关系很重要的情况下，例如零售公司不仅使用历史数据，还使用有关客户相似性和在线行为的上下文数据进行个性化的产品推荐。虚拟智能助手 Alexa 使用了多层上下文模型来提高准确率。2018 年，Alexa 还拥有了上下文接转功能（context carryover），在回答新问题时可以将之前的参考资料纳入对话中。

然而现在许多机器学习方法忽视了丰富的上下文信息，这是因为机器学习依赖通过元组构建的输入数据，而忽略了许多预测关系和网络化的数据。此外，上下文信息的获取并不总是那么容易，也许很难访问和处理。对于传统方法来说，查找 4 跳甚至更多跳数之外的联系也是一大挑战。借助图，即可轻松获取和整合那些相互关联的数据。

8.2　关联特征提取与特征选择

特征提取和特征选择可用于获取原始数据，并为训练机器学习模型创建合适的子集和格式。这是一个基础步骤，执行得当的话，机器学习的预测结果会更一致、更准确。

特征提取与特征选择

特征提取（feature extraction）用于将大量数据和属性提取为一组具有代表性的描述性属性。该过程针对输入数据的个性特征或模式获取数值（特征），以便在其他数据中区分类别。当模型难以直接分析数据时（可能由于规模、格式或偶然需要进行比较），就会用到特征提取。

特征选择（feature selection）是在已提取特征中确定对定向目标最重要或影响最大的子集的过程。它用于揭示预测的重要程度和效率。假设有 20 个特征，其中 13 个特征加起来能够解释 92% 需要预测的内容，那么可以在模型中去除其他 7 个特征。

特征的正确组合有助于提高准确率，这是因为它从根本上影响模型的学习方式。因为即使很小的改进也会造成显著差别，所以本章重点介绍**关联特征**（connected feature）。关联特征是根据数据结构提取的特征。这些特征既可以通过局部图查询得到（某个节点周围的局部图），也可以通过全局图查询得到（基于关联特征提取所用的关系，使用图算法在数据中识别预测性元素）。

重要的不仅是要得到正确的特征组合，还要去除不必要的特征，以防止模型的针对性过强。这样做可以避免创建仅对训练数据有效的模型（称为**过拟合**），而且可以显著扩展适用性。还可以使用图算法来评估这些特征，并确定哪些特征对模型的关联特征选择影响最大。例如可以将特征映射为图的节点，基于相似特征创建关系，然后计算特征的中心度。特征关系可以根据保持簇的数据点密度的能力来定义。Khadidja Henni、Neila Mezghani 和 Charles Gouin-Vallerand 在论文“Unsupervised Graph-Based Feature Selection Via Subspace and PageRank Centrality”中介绍了该方法，他们采用了高维度、小规模样本的数据集。

图嵌入

图嵌入（graph embedding）是图中节点和关系的**特征向量**（feature vector）表示。特征向量只不过是经过维度映射的特征集合，如图 8-2 所示的 (x, y, z) 坐标。

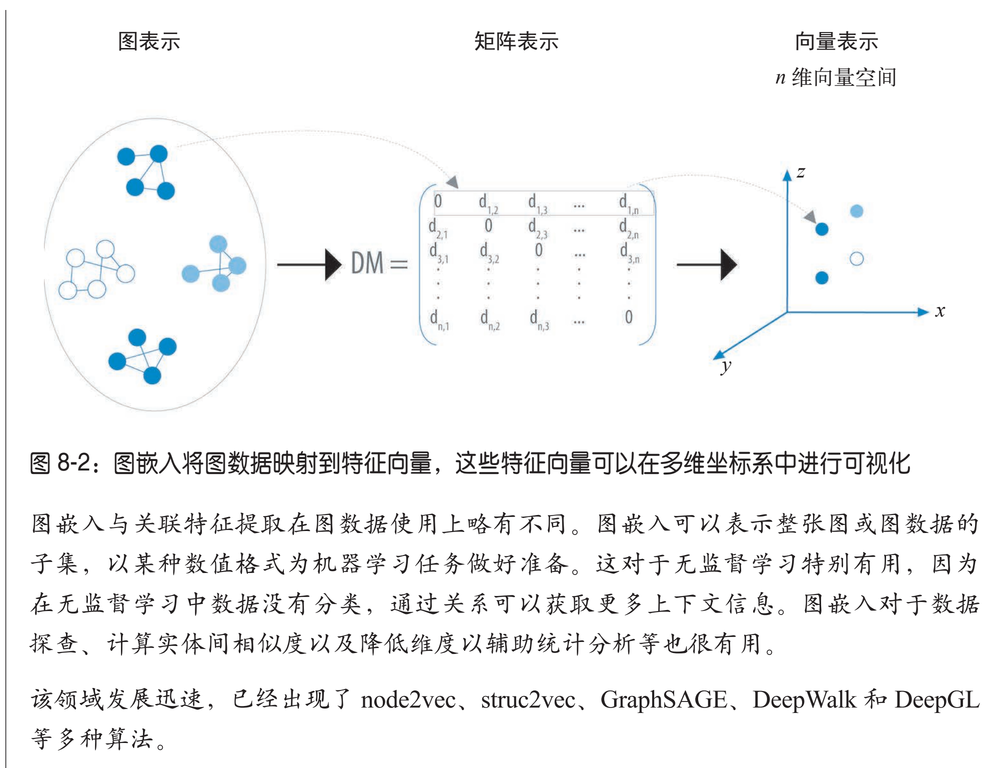


图 8-2：图嵌入将图数据映射到特征向量，这些特征向量可以在多维坐标系中进行可视化

图嵌入与关联特征提取在图数据使用上略有不同。图嵌入可以表示整张图或图数据的子集，以某种数值格式为机器学习任务做好准备。这对于无监督学习特别有用，因为在无监督学习中数据没有分类，通过关系可以获取更多上下文信息。图嵌入对于数据探查、计算实体间相似度以及降低维度以辅助统计分析等也很有用。

该领域发展迅速，已经出现了 node2vec、struc2vec、GraphSAGE、DeepWalk 和 DeepGL 等多种算法。

下面介绍关联特征的类型及其用法。

8.2.1 图特征

图特征（graphy feature）是指任意数量与图的关联性相关的度量指标，例如进出节点的关系数量、隐含的三角形数量和共同邻节点的数量等。本章将通过这些指标研究示例，因为它们易于收集，并且能够很好地验证早期假设。

此外，如果明确知道目标，那么可以使用特征工程，例如想知道有多少人拥有 4 跳以上的欺诈账户。这种方法使用图遍历可以非常高效地查找关系的深层路径，查看标签、属性、数量和推断关系。

上述过程很容易自动化，可以将这些预测性图特征提交到已有管道中，例如可以抽象诈骗分子关系数量，并将该数量添加为节点属性，以便用于其他机器学习任务。

8.2.2 图算法特征

如果知道所要寻找的一般结构但不知道确切模式，还可以使用图算法来寻找特征。假设我们发现某些类型的社团分组存在欺诈迹象，也许其中存在某种典型的关系密度或层级结

构。在这种情况下，并不需要某个确切组织的严格特征，而只要一种灵活且全局相关的结构。本章示例将使用社团发现算法来提取关联特征，也会经常用到像 PageRank 算法这样的中心性算法。

此外，组合使用多种关联特征的方法往往优于仅使用单一关联特征的方法。例如可以将关联特征与通过 Louvain 模块度算法发现的社团、使用 PageRank 算法计算的有影响力的节点，以及已知诈骗分子 3 跳之外的度量指标综合运用来预测诈骗。

图 8-3 展示了一种组合方法，其作者将 PageRank 算法和着色算法等图算法与入度和出度等图的度量指标组合使用。该图摘自 Shobeir Fakhraei 等人的论文"Collective Spammer Detection in Evolving Multi-Relational Social Networks"。

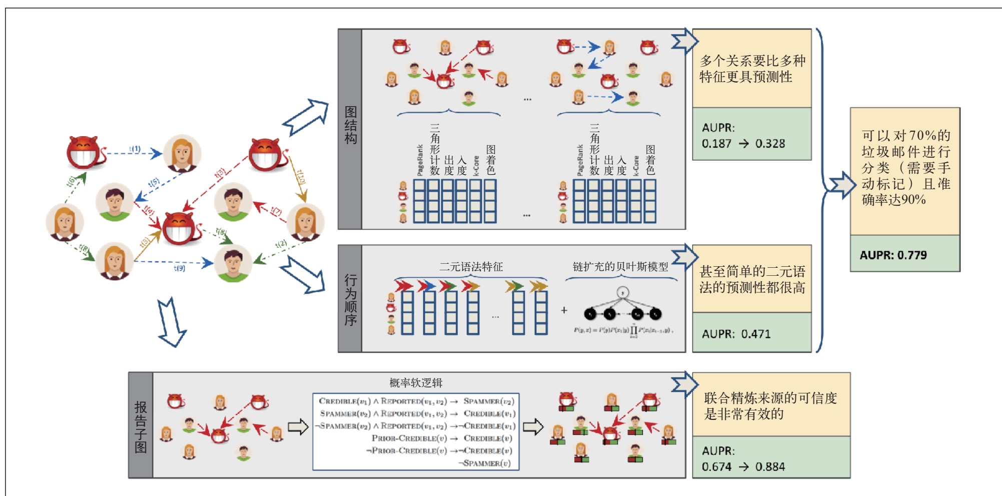


图 8-3：关联特征提取可以与其他预测方法组合使用以改进预测结果。AUPR 是精确率 - 召回率曲线之下的面积，数值越大越好

图结构部分展示了如何使用几种图算法提取关联特征。有趣的是，作者发现从多种关系提取关联特征比简单添加更多特征参数更具预测性。报告子图部分展示了如何将图特征转换为机器学习模型可用的特征。通过在图增强机器学习工作流程中组合使用多种方法，作者改进了之前的检测方法，可对 70% 的垃圾邮件（需要预先手动标记）进行分类，准确率达 90%。

如果已经提取了关联特征，还可以通过像 PageRank 算法这样的图算法来改进训练，以便按照影响力划分特征的优先级。这样做能够充分表示数据，同时消除可能导致结果变差或处理速度变慢的噪声变量。有了这类信息，还可以通过特征约简来识别高共现率的特征，以便进一步对模型调优。Dino Ienco、Rosa Meo 和 Marco Botta 在论文"Using PageRank in Feature Selection"中介绍了这种方法。

前面讨论了如何将关联特征应用于涉及欺诈检测和垃圾邮件检测的场景。对这些情况，行

为通常隐藏在多重迷雾和网络关系中。如果没有图承载的上下文信息，仅靠传统的特征提取和选择方法，可能无法检测到这些行为。

关联特征增强机器学习的另一个领域（也是本章后续内容的重点）是**链接预测**（link prediction）。链接预测可用于估计未来形成某种关系的可能性，或者说该关系可能已经存在于图中，只是由于数据不完整而丢失了。由于网络是动态的并且可以快速增长，因此对即将添加的链接进行预测有广泛的用途，例如产品推荐、药物重定向，甚至推断犯罪关系等。

从图中提取的关联特征通常用于改进链接预测结果，这包括基本的图特征以及通过中心性算法和社团发现算法提取的特征。基于节点接近度或相似度的链接预测也是标准方法，David Liben-Nowell 和 Jon Kleinberg 在论文“The Link Prediction Problem for Social Networks”中提出，在发现节点的接近度方面，仅网络结构本身就可能包含足够的潜在信息，而且优于更直接的度量方法。

通过关联特征增强机器学习的方法已介绍完毕，下面深入研究链接预测示例，看看如何应用图算法来改进预测结果。

8.3 图与机器学习实践：链接预测

接下来展示一个基于引文网络数据集的实用示例，该数据集是从 DBLP、ACM 和 MAG 提取的研究数据集。唐杰等人撰写的论文“ArnetMiner: Extraction and Mining of Academic Social Networks”介绍了该数据集，其最新版本包含 3 079 007 篇论文、1 766 547 位作者、9 437 718 个作者关系，以及 25 166 994 个引用关系。

我们将采用该数据集的一个子集，其中出现的论文主要来自以下出版物：

- *Lecture Notes in Computer Science*
- *Communications of the ACM*
- *International Conference on Software Engineering*
- *Advances in Computing and Communications*

所得数据集包含 51 956 篇论文、80 299 位作者、140 575 个作者关系，以及 28 706 个引用关系。可以基于有过论文合著关系的作者创建合著者关系图，然后预测几对作者未来有无可能合作。我们只对此前未合作过的作者之间可能存在合作关系感兴趣，并不关心一对作者之间的多个合作关系。

首先安装所需工具并将数据导入 Neo4j，然后介绍如何正确均衡数据，将样本分割为 Spark 用于训练和测试的 DataFrame 对象，还将介绍我们的假设和链接预测方法，并在 Spark 中创建一个机器学习管道，最后示范如何训练和评估各种预测模型，起初采用基本的图特征，之后添加更多使用 Neo4j 提取的图算法特征。

8.3.1 工具和数据

首先安装工具并导入数据，然后探查数据集并创建一个机器学习管道。

做其他工作之前，先安装要用到的库。

- py2neo

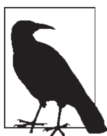
  一个与 Python 数据科学生态系统很好集成的 Neo4j Python 库。

- pandas

  一个用于整理数据库外部数据的高性能库，具有易用的数据结构和数据分析工具。

- Spark MLlib

  Spark 的机器学习库。



我们使用 MLlib 作为机器学习库的示例。本章所示方法可与其他机器学习库（如 scikit-learn）联用。

所有代码都将在 PySpark REPL 中运行。可以通过以下命令启动 REPL：

```
export SPARK_VERSION="spark-2.4.0-bin-hadoop2.7"
./${SPARK_VERSION}/bin/pyspark \
  --driver-memory 2g \
  --executor-memory 6g \
  --packages julioasotodv:spark-tree-plotting:0.2
```

这与第 3 章使用的 REPL 启动命令类似，只是没有加载 GraphFrames 库，而是加载了 spark-tree-plotting 包。在编写本书之时，Spark 的最新发布版本是 spark-2.4.0-bin-hadoop2.7。当你阅读本书时，版本可能已经发生变化，若是如此，请确保正确更改 `SPARK_VERSION` 环境变量。

启动 REPL 后，可以导入要用到的库，如下所示：

```
from py2neo import Graph
import pandas as pd
from numpy.random import randint

from pyspark.ml import Pipeline
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import StringIndexer, VectorAssembler
from pyspark.ml.evaluation import BinaryClassificationEvaluator

from pyspark.sql.types import *
from pyspark.sql import functions as F
```

```
from sklearn.metrics import roc_curve, auc
from collections import Counter

from cycler import cycler
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
```

然后创建到 Neo4j 数据库的连接。

```
graph = Graph("bolt://localhost:7687", auth=("neo4j", "neo"))
```

8.3.2 将数据导入Neo4j

下面将数据加载到 Neo4j 中，同时均衡分割训练数据和测试数据。需要下载该数据集第 10 个版本的 ZIP 文件，解压并将其中内容放入 import 文件夹中。现有以下文件：

- dblp-ref-0.json
- dblp-ref-1.json
- dblp-ref-2.json
- dblp-ref-3.json

有了 import 文件夹中的这些文件，还需要在 Neo4j 配置文件中加入下述性质，以便使用 APOC 库来处理数据。

```
apoc.import.file.enabled=true
apoc.import.file.use_neo4j_config=true
```

首先创建约束以确保论文或作者不会重复：

```
CREATE CONSTRAINT ON (article:Article)
ASSERT article.index IS UNIQUE;

CREATE CONSTRAINT ON (author:Author)
ASSERT author.name IS UNIQUE;
```

然后执行下述查询，从 JSON 文件中导入数据：

```
CALL apoc.periodic.iterate(
  'UNWIND ["dblp-ref-0.json","dblp-ref-1.json",
           "dblp-ref-2.json","dblp-ref-3.json"] AS file
   CALL apoc.load.json("file:///" + file)
   YIELD value
   WHERE value.venue IN ["Lecture Notes in Computer Science",
                         "Communications of The ACM",
                         "international conference on software engineering",
                         "advances in computing and communications"]
   return value',
  'MERGE (a:Article {index:value.id})
```

```
    ON CREATE SET a += apoc.map.clean(value,["id","authors","references"],[0])
    WITH a,value.authors as authors
    UNWIND authors as author
    MERGE (b:Author{name:author})
    MERGE (b)<-[:AUTHOR]-(a)'
, {batchSize: 10000, iterateList: true});
```

这段代码将产生图 8-4 所示的图模式。

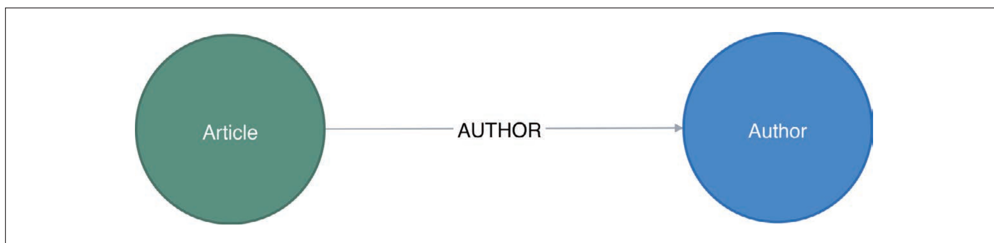


图 8-4：引用图

由于连接论文和作者的图还比较简单，因此要添加更多从关系中推断出的信息来辅助预测。

8.3.3　合著者关系图

要预测作者之间未来的合作，首先要创建合著者关系图。下面的 Neo4j Cypher 查询将为合著论文的每对作者创建 `CO_AUTHOR` 关系。

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
MERGE (a1)-[coauthor:CO_AUTHOR {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

我们在查询中设置 `CO_AUTHOR` 关系的 `year` 性质，它表示这两位作者最初合作的年份。我们只对两位作者的首次合作感兴趣，而暂时不关心后续合作。

图 8-5 是所创建图的一部分示例，它显示出一些有意思的社团结构。图中的每个圆表示一位作者，圆之间的连线表示 `CO_AUTHOR` 关系，因此图的左边是 4 位相互有合作关系的作者，而右边的两个例子分别描述了 3 位作者之间的合作关系。加载完数据和基本图后，开始创建用于训练和测试的两个数据集。

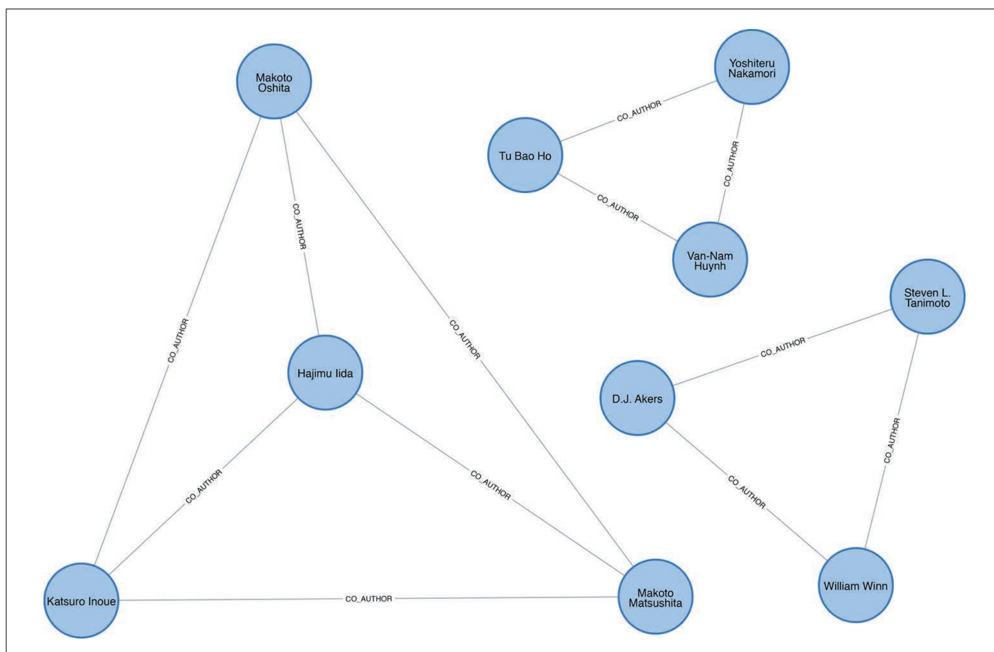


图 8-5：合著者关系图

8.3.4 创建均衡的训练数据集和测试数据集

链接预测问题旨在尝试并预测未来的链接创建。该数据集适用于这项工作，这是因为可以根据论文的日期来分割数据。需要判定该用哪一年来确定训练与测试的分割点，用那一年之前的数据对模型进行全方位训练，用那一年之后的数据测试所创建的链接。

先看看这些论文发表的时间。可以编写以下查询按年份分组论文并统计数量：

```
query = """
MATCH (article:Article)
RETURN article.year AS year, count(*) AS count
ORDER BY year
"""

by_year = graph.run(query).to_data_frame()
```

通过条形图来可视化查询结果，代码如下：

```
plt.style.use('fivethirtyeight')
ax = by_year.plot(kind='bar', x='year', y='count', legend=None, figsize=(15,8))
ax.xaxis.set_label_text("")
plt.tight_layout()
plt.show()
```

运行代码，可以得到如图 8-6 所示的条形图。

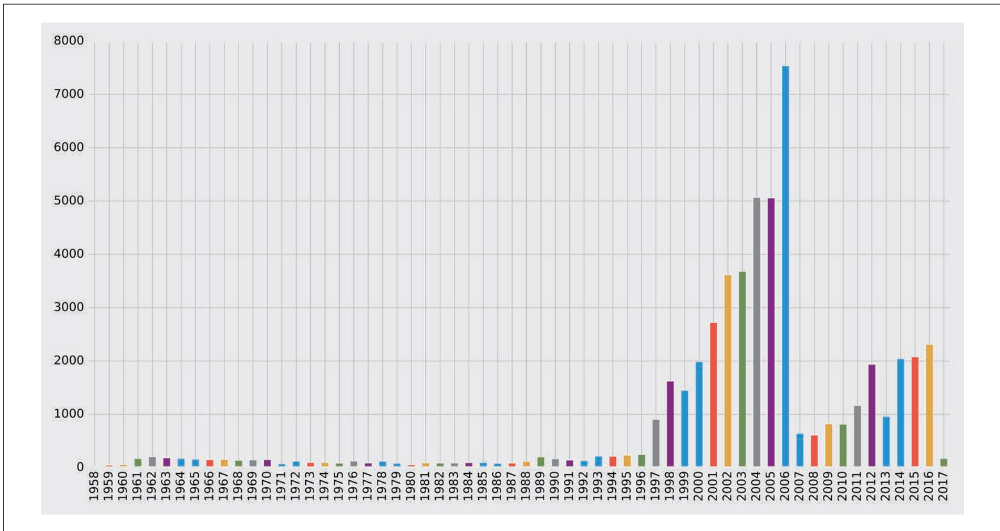


图 8-6：按年份统计论文

1997 年之前发表的论文很少，2001 年至 2006 年发表的论文很多。在 2011 年之前的几年，论文数量有一些下降，此后又逐渐上升（不包括 2013 年）。看来 2006 年不错，可以据此分割数据以训练模型并做出预测。来看看 2006 年之前发表的论文数量，以及该年度和之后发表的论文数量。通过下述查询进行计算：

```
MATCH (article:Article)
RETURN article.year < 2006 AS training, count(*) AS count
```

结果如下所示，其中 `true` 表示论文发表于 2006 年之前。

training	count
false	21059
true	30897

结果不错！ 60% 的论文是在 2006 年之前发表的，而 2006 年及之后发表的论文占 40%。这对训练和测试而言是相当均衡的数据分割了。

同样，使用 2006 年来分割作者合作关系。为首次合作发生在 2006 年之前的每对作者创建 `CO_AUTHOR_EARLY` 关系。

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year < 2006
```

```
MERGE (a1)-[coauthor:CO_AUTHOR_EARLY {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

然后为首次合作发生在 2006 年及之后的每对作者创建 CO_AUTHOR_LATE 关系。

```
MATCH (a1)<-[:AUTHOR]-(paper)-[:AUTHOR]->(a2:Author)
WITH a1, a2, paper
ORDER BY a1, paper.year
WITH a1, a2, collect(paper)[0].year AS year, count(*) AS collaborations
WHERE year >= 2006
MERGE (a1)-[coauthor:CO_AUTHOR_LATE {year: year}]-(a2)
SET coauthor.collaborations = collaborations;
```

在构建训练集和测试集之前，先核对存在链接关系的节点对数量。下述查询将返回存在 CO_AUTHOR_EARLY 关系的节点对数量。

```
MATCH ()-[:CO_AUTHOR_EARLY]->()
RETURN count(*) AS count
```

执行该查询，结果如下所示：

count
81096

以下查询将查找存在 CO_AUTHOR_LATE 关系的节点对数量。

```
MATCH ()-[:CO_AUTHOR_LATE]->()
RETURN count(*) AS count
```

执行该查询，结果如下所示：

count
74128

下面准备构建训练数据集和测试数据集。

均衡分割数据

具有 CO_AUTHOR_EARLY 关系和 CO_AUTHOR_LATE 关系的节点对可以作为正例，还需要创建一些反例。现实世界中的大多数网络是稀疏的，关系相对集中，这张图也并无不同。两个节点间没有关系的示例数量远远多于有关系的示例数量。

如果查询 CO_AUTHOR_EARLY 数据，会发现 45 018 位作者具有这种关系，但是这些作者之间只有 81 096 个关系。听起来可能并没有那么不均衡，但实际上确实如此：我们的图可能拥有的最大关系数是 $(45\ 018 \times 45\ 017) / 2 = 1\ 013\ 287\ 653$，这意味着有很多反例（没有链接的情况）。如果使用所有反例来训练模型，会出现严重的类别失衡问题。通过预测每对节点间不存在关系，模型可以达到极高的准确度。

Ryan Lichtenwalter、Jake Lussier 和 Nitesh Chawla 在论文“New Perspectives and Methods in Link Prediction”中介绍了应对这一挑战的几种方法。其中一种方法是在邻域中寻找目前尚未连接的节点以构建反例。

我们通过寻找彼此距离两三跳的节点对来构建反例，这不包括那些已经存在关系的节点对。然后，对这些节点对进行下采样，从而得到相同数量的正例和反例。



彼此相距 2 跳且不存在关系的节点对共有 314 248 个。如果将距离增加到 3 跳，则有 967 677 个节点对。

可用以下函数来下采样反例：

```
def down_sample(df):
    copy = df.copy()
    zero = Counter(copy.label.values)[0]
    un = Counter(copy.label.values)[1]
    n = zero - un
    copy = copy.drop(copy[copy.label == 0].sample(n=n, random_state=1).index)
    return copy.sample(frac=1)
```

该函数计算正例和反例数目之差，然后抽样反例，使正反例的数目相等。可以运行以下代码来构建一个正例和反例均衡的训练集。

```
train_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_EARLY]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

train_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_EARLY]-()
MATCH (author)-[:CO_AUTHOR_EARLY*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR_EARLY]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

train_missing_links = train_missing_links.drop_duplicates()
training_df = train_missing_links.append(train_existing_links, ignore_index=True)
training_df['label'] = training_df['label'].astype('category')
training_df = down_sample(training_df)
training_data = spark.createDataFrame(training_df)
```

现在已经强制将 `label` 列作为一个类别了，其中，`1` 代表节点对之间存在链接，而 `0` 代表节点对之间没有链接。运行以下代码可以在 DataFrame 对象中观察这些数据，结果如下所示：

```
training_data.show(n=5)
```

node1	node2	label
10019	28091	1
10170	51476	1
10259	17140	0
10259	26047	1
10293	71349	1

该结果显示了一个节点对列表以及节点间是否存在合著关系。例如节点 10019 和 28091 的 label 值为 1，这表示二者之间存在合著关系。

运行以下代码来检查 DataFrame 对象的内容：

```
training_data.groupby("label").count().show()
```

结果如下所示：

label	count
0	81096
1	81096

已经创建了正例和反例数目相同的训练集，下面以同样的方式构建测试集。以下代码将构建一个正例和反例均衡的测试集。

```
test_existing_links = graph.run("""
MATCH (author:Author)-[:CO_AUTHOR_LATE]->(other:Author)
RETURN id(author) AS node1, id(other) AS node2, 1 AS label
""").to_data_frame()

test_missing_links = graph.run("""
MATCH (author:Author)
WHERE (author)-[:CO_AUTHOR_LATE]-()
MATCH (author)-[:CO_AUTHOR*2..3]-(other)
WHERE not((author)-[:CO_AUTHOR]-(other))
RETURN id(author) AS node1, id(other) AS node2, 0 AS label
""").to_data_frame()

test_missing_links = test_missing_links.drop_duplicates()
test_df = test_missing_links.append(test_existing_links, ignore_index=True)
test_df['label'] = test_df['label'].astype('category')
test_df = down_sample(test_df)
test_data = spark.createDataFrame(test_df)
```

运行以下代码来检查 DataFrame 对象的内容：

```
test_data.groupby("label").count().show()
```

结果如下所示：

label	count
0	74128
1	74128

有了均衡的训练数据集和测试数据集，下面开始研究预测链接的方法。

8.3.5 如何预测缺失链接

从一些基本假设开始，即数据中的哪些元素可以预测两位作者以后是否会成为合著者。假设因领域和问题而异，但在本例中，最具预测性的特征与社团有关。刚开始假设下述要素可增加作者成为合著者的可能性：

- 有较多的共同作者；
- 作者之间存在潜在的三元关系；
- 作者有较多关系；
- 作者都在同一社团；
- 作者都在同一联系较紧密的社团。

我们将基于上述假设构建图特征，并使用这些特征来训练二元分类器。二元分类是机器学习的一种，其任务是根据规则预测一个元素属于两个预定义群组中的哪一个。我们使用分类器基于分类规则预测一对作者之间是否有链接。对于本例，值 `1` 表示有链接（有合著关系），值 `0` 表示没有链接（无合著关系）。

在 Spark 中可以用随机森林来实现二元分类器。**随机森林**（random forest）是一种用于分类、回归和其他任务的集成学习方法，如图 8-7 所示。

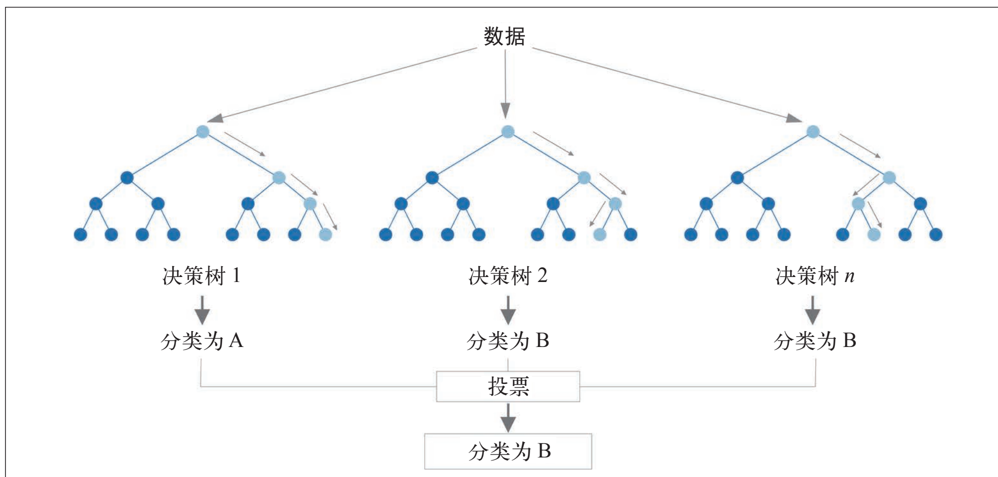


图 8-7：随机森林构建了一个决策树集合，然后依据多数投票（分类问题）或平均值（回归问题）得出结果

随机森林分类器从训练好的多棵决策树中提取结果，然后通过投票来预测分类——在本例中，无论是否存在链接（合著关系），都是如此。

下面创建工作流程。

8.3.6　创建机器学习管道

我们将在 Spark 中创建基于随机森林分类器的机器学习管道。由于所用数据集由强特征和弱特征混合构成，因此这种方法非常适用。弱特征有时很有用，随机森林方法所创建的模型并非仅适用于这里的训练数据。

要创建机器学习管道，可用 `fields` 变量传递特征列表（分类器会用到）。分类器用名为 `features` 的列来单独接收这些特征参数，因此还要使用 `VectorAssembler` 将数据转换成所需格式。

以下代码创建了一个机器学习管道，并且使用 MLlib 设置参数：

```
def create_pipeline(fields):
    assembler = VectorAssembler(inputCols=fields, outputCol="features")
    rf = RandomForestClassifier(labelCol="label", featuresCol="features",
                                numTrees=30, maxDepth=10)
    return Pipeline(stages=[assembler, rf])
```

`RandomForestClassifier` 函数用到了下述参数。

- `labelCol`
  包含待预测变量的字段名，这里指一对节点是否存在链接。
- `featuresCol`
  字段名，包含用于预测一对节点是否存在链接的变量。
- `numTrees`
  构成随机森林的决策树的数量。
- `maxDepth`
  决策树的最大深度。

我们基于实验来选择决策树的数量和深度。可以考虑采用超参数，如可对算法进行性能调优的设置。优良的超参数通常很难预先确定，调整模型通常需要经历一些试错。

介绍过了基础知识并创建了管道，下面开始创建模型并评估其表现。

8.3.7 预测链接：基本图特征

我们将从创建简单模型开始，基于从共同作者、择优连接和邻节点并集总数提取的特征，尝试预测两位作者未来是否会合作。

- **共同作者**

  查找两位作者之间的潜在三角形数。这主要基于这样的理念：有共同作者的两位作者将来可能会被相互引见并开展协作。

- **择优连接**

  将每对作者的合著者数量相乘，为每对作者生成一个分值。按照常理，一位作者更有可能与已经合著了大量论文的人合作。

- **邻节点并集总数**

  找到每位作者的合著者总数，然后减去重复数。

在 Neo4j 中，可以使用 Cypher 查询来计算这些值。下面的函数将计算这些针对训练集的度量指标：

```
def apply_graphy_training_features(data):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           size([(p1)-[:CO_AUTHOR_EARLY]-(a)-
                      [:CO_AUTHOR_EARLY]-(p2) | a]) AS commonAuthors,
           size((p1)-[:CO_AUTHOR_EARLY]-()) * size((p2)-
                     [:CO_AUTHOR_EARLY]-()) AS prefAttachment,
           size(apoc.coll.toSet(
             [(p1)-[:CO_AUTHOR_EARLY]-(a) | id(a)] +
                   [(p2)-[:CO_AUTHOR_EARLY]-(a) | id(a)]
           )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                                for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
                                {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

下面的函数将针对测试集来计算这些值：

```
def apply_graphy_test_features(data):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
```

```
            pair.node2 AS node2,
            size([(p1)-[:CO_AUTHOR]-(a)-[:CO_AUTHOR]-(p2) | a]) AS commonAuthors,
            size((p1)-[:CO_AUTHOR]-()) * size((p2)-[:CO_AUTHOR]-())
                                    AS prefAttachment,
            size(apoc.coll.toSet(
              [(p1)-[:CO_AUTHOR]-(a) | id(a)] + [(p2)-[:CO_AUTHOR]-(a) | id(a)]
            )) AS totalNeighbors
    """
    pairs = [{"node1": row["node1"], "node2": row["node2"]}
                       for row in data.collect()]
    features = spark.createDataFrame(graph.run(query,
                       {"pairs": pairs}).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

这两个函数都以 DataFrame 对象为参数，该对象包含 `node1` 列和 `node2` 列中的节点对；然后构建包含这些节点对的映射数组，并为每对节点计算各种度量指标。



在本章中，`UNWIND` 子句对于获取大规模节点对集合并在查询中返回其所有特征非常有用。

以下代码可以把 Spark 中的函数应用于训练集 DataFrame 对象和测试集 DataFrame 对象：

```
training_data = apply_graphy_training_features(training_data)
test_data = apply_graphy_test_features(test_data)
```

探查一下训练集中的数据。以下代码绘制了关于 `commonAuthors` 出现频率的条形图，如图 8-8 所示：

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    common_authors = filtered.toPandas()["commonAuthors"]
    histogram = common_authors.value_counts().sort_index()
    histogram /= float(histogram.sum())
    histogram.plot(kind="bar", x='Common Authors', color="darkblue",
                   ax=axs[index], title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Common Authors")

plt.tight_layout()
plt.show()
```

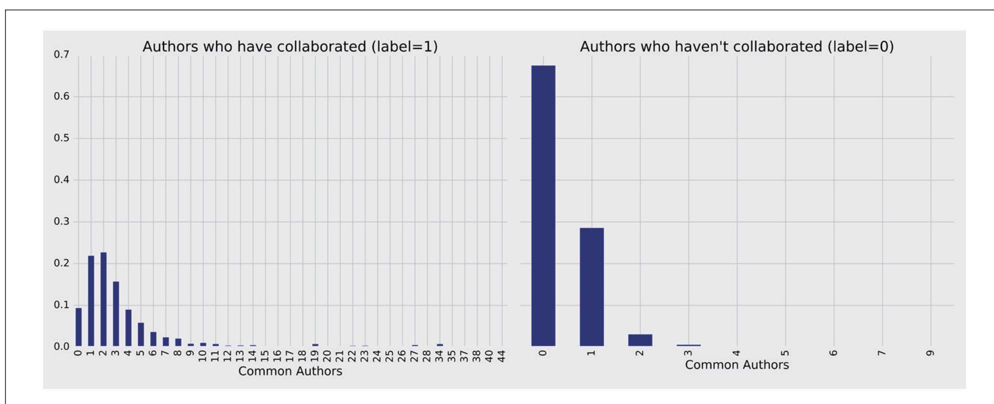


图 8-8：commonAuthors 出现频率

左边是当作者之间有过合作时 commonAuthors 的频率，右边是作者之间没合作过时 commonAuthors 的频率。对于那些没有合作过的作者（图 8-8 右侧），最大共同作者数是 9，但是 95% 的人的值是 1 或 0。在没有合著过论文的人当中，共同作者数基本也不多，这并不奇怪。对于那些合作过的人（图 8-8 左侧），70% 的人共同合作者少于 5 位，有一两位合作者的人最多。

现在训练模型来预测缺失链接。下面的函数可完成该操作：

```
def train_model(fields, training_data):
    pipeline = create_pipeline(fields)
    model = pipeline.fit(training_data)
    return model
```

刚开始创建一个仅采用 commonAuthors 的基本模型，代码如下：

```
basic_model = train_model(["commonAuthors"], training_data)
```

模型训练完毕后，看看该模型针对虚拟数据的表现。下面的代码针对不同的 commonAuthors 取值求解预测值：

```
eval_df = spark.createDataFrame(
    [(0,), (1,), (2,), (10,), (100,)],
    ['commonAuthors'])

(basic_model.transform(eval_df)
 .select("commonAuthors", "probability", "prediction")
 .show(truncate=False))
```

运行以上代码，结果如下所示：

commonAuthors	probability	prediction
0	[0.7540494940434322,0.24595050595656787]	0.0
1	[0.7540494940434322,0.24595050595656787]	0.0
2	[0.0536835525078107,0.9463164474921892]	1.0
10	[0.0536835525078107,0.9463164474921892]	1.0

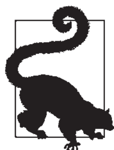
如果 `commonAuthors` 值小于 2，则作者之间没有合作关系的概率为 75%，所以模型预测值为 0。如果 `commonAuthors` 值大于或等于 2，则作者之间有合作关系的概率为 94%，所以模型预测值为 1。

下面使用测试集来评估模型。尽管评估模型运行情况的方法有多种，但是大多数方法源自一些基准预测指标，如表 8-1 所示。

表8-1：预测指标

指　标	公　式	描　述
准确率	$\frac{正例数 + 负例数}{预测总数}$	模型正确预测的分值，或者说正确预测数除以预测总数。请注意，这样的准确率本身就可能产生误导，当数据不均衡时尤甚。如果有一个数据集包含 95 只猫和 5 只狗，那么当模型预测每幅图像都是猫时，尽管一只狗都没有正确识别出来，但准确率仍达 95%
精确率	$\frac{正例数}{正例数 + 假正例数}$	正确识别正例的比例。较低的精确率表明出现较多假正例。模型未出现假正例则精确率为 100%
召回率（正例率）	$\frac{正例数}{正例数 + 假负例数}$	实际正确识别正例的比例。较低的召回率表示假负例较多。模型没有产生假负例则其召回率为 100%
假正例率	$\frac{假正例数}{假正例数 + 负例数}$	识别不正确正例的比例。较高分值表示有较多的假正例
受试者工作特征曲线（ROC 曲线）	*X-Y* 图表	ROC 曲线是召回率（正例率）与假正例率在不同分类阈值上的曲线。曲线下的面积（AUC）表示 ROC 曲线下从 *X-Y* 轴（0, 0）到（1, 1）的面积

我们将使用准确率、精确率、召回率和 ROC 曲线来评估模型。准确率是一个粗略指标，因此重点是要提高整体精确率和召回率。我们将使用 ROC 曲线来比较单个特征对预测率的影响。



目标不同，可能希望采取的指标也不同，例如可能希望消除疾病指标的所有假负例，但并不希望所有预测都是阳性结果。我们可能会为不同模型设置多个阈值，将结果传递给二次检查，看是否可能出现假性结果。

降低分类阈值会导致更全面的阳性结果，同时增加假正例和正例。

可使用以下函数来计算这些预测指标：

```
def evaluate_model(model, test_data):
    # 针对测试集运行模型
    predictions = model.transform(test_data)

    # 计算正例数、假正例数以及假负例数
    tp = predictions[(predictions.label == 1) &
                     (predictions.prediction == 1)].count()
    fp = predictions[(predictions.label == 0) &
                     (predictions.prediction == 1)].count()
    fn = predictions[(predictions.label == 1) &
                     (predictions.prediction == 0)].count()

    # 手动计算召回率和精确率
    recall = float(tp) / (tp + fn)
    precision = float(tp) / (tp + fp)

    # 使用Spark MLlib的二元分类求解程序计算准确率
    accuracy = BinaryClassificationEvaluator().evaluate(predictions)

    # 使用scikit-learn中的函数计算假正例率和正例率
    labels = [row["label"] for row in predictions.select("label").collect()]
    preds = [row["probability"][1] for row in predictions.select
                ("probability").collect()]
    fpr, tpr, threshold = roc_curve(labels, preds)
    roc_auc = auc(fpr, tpr)

    return { "fpr": fpr, "tpr": tpr, "roc_auc": roc_auc, "accuracy": accuracy,
             "recall": recall, "precision": precision }
```

然后编写函数来以易用的格式展现结果：

```
def display_results(results):
    results = {k: v for k, v in results.items() if k not in
                    ["fpr", "tpr", "roc_auc"]}
    return pd.DataFrame({"Measure": list(results.keys()),
                         "Score": list(results.values())})
```

用以下代码调用该函数并且显示结果：

```
basic_results = evaluate_model(basic_model, test_data)
display_results(basic_results)
```

共同作者模型的预测指标如下所示：

measure	score
accuracy	0.864457
recall	0.753278
precision	0.968670

效果不错，因为对未来合作的预测仅基于作者对中的共同作者的数量；然而，如果综合考虑多个指标，就会看得更清楚。例如该模型的精确率为 0.968670，这意味着它很好地预测了链接的**存在**；然而其召回率为 0.753278，这意味着它并不擅长预测链接**不存在**的情况。

可以用以下函数绘制 ROC 曲线（正例与假正例的相关性）：

```
def create_roc_plot():
    plt.style.use('classic')
    fig = plt.figure(figsize=(13, 8))
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.rc('axes', prop_cycle=(cycler('color',
                   ['r', 'g', 'b', 'c', 'm', 'y', 'k'])))
    plt.plot([0, 1], [0, 1], linestyle='--', label='Random score
                   (AUC = 0.50)')
    return plt, fig


def add_curve(plt, title, fpr, tpr, roc):
    plt.plot(fpr, tpr, label=f"{title} (AUC = {roc:0.2})")
```

可按如下方式调用该函数：

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

图 8-9 展示了基本模型的 ROC 曲线。共同作者模型的曲线下面积（AUC）分值为 0.86。虽然这是一个全面的预测指标，但是需要用图表（或其他指标）来评估是否符合预期目标。图 8-9 显示，当正例率（召回率）接近 80% 时，假正例率达到 20% 左右。这对于欺诈检测这样的场景可能会有问题，因为追踪假正例的代价很大。

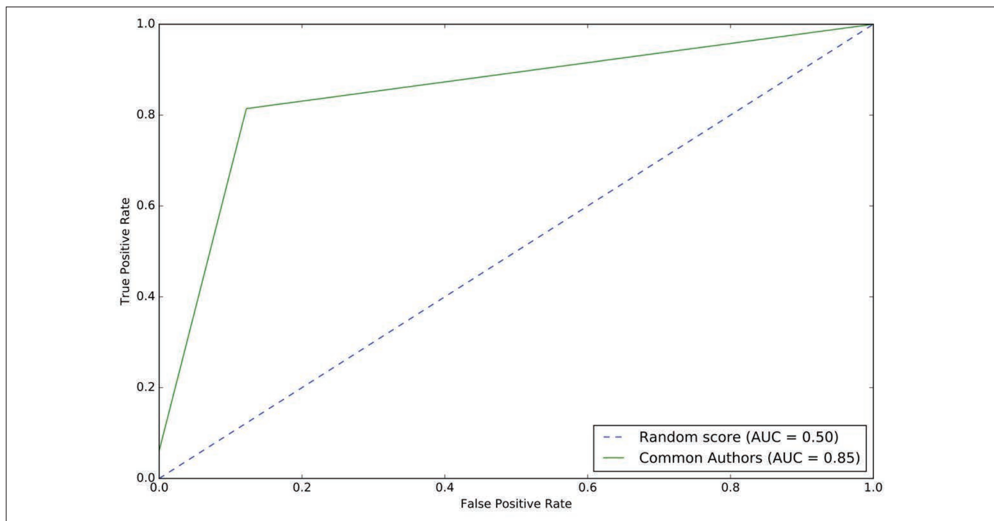


图 8-9：基本模型的 ROC 曲线

下面使用其他图特征来改进预测结果。在训练模型之前，先看看数据的分布情况。可以运行以下代码来显示每种图特征的描述性统计数据：

```
(training_data.filter(training_data["label"]==1)
 .describe()
 .select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
 .show())

(training_data.filter(training_data["label"]==0)
 .describe()
 .select("summary", "commonAuthors", "prefAttachment", "totalNeighbors")
 .show())
```

运行这段代码，结果如下所示：

summary	commonAuthors	prefAttachment	totalNeighbors
count	81096	81096	81096
mean	3.5959233501035808	69.93537289138798	10.082408503502021
stddev	4.715942231635516	171.47092255919472	8.44109970920685
min	0	1	2
max	44	3150	90

summary	commonAuthors	prefAttachment	totalNeighbors
count	81096	81096	81096
mean	0.37666469369635985	48.18137762651672	12.97586810693499
stddev	0.6194576095461857	94.92635344980489	10.082991078685803
min	0	1	1
max	9	1849	89

链接（有合著关系）和无链接（无合著关系）之间有较大区别的特征可能更具预测性，这是因为其差异更大。与没有合作过的作者相比，合作过的作者的 prefAttachment 平均值更高。对于 commonAuthors 来说，这种差异甚至更显著。可以看到 totalNeighbors 的值没有太大差异，这可能意味着该特征的预测性不强。同样值得关注的是较大的标准偏差以及择优连接的最小值和最大值。这正是我们所期望的聚焦中心（有“超级连接者”）的小世界网络。

运行以下代码来训练新模型，添加择优连接和邻节点并集总数两个特征：

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors"]
graphy_model = train_model(fields, training_data)
```

然后求解模型并展示结果：

```
graphy_results = evaluate_model(graphy_model, test_data)
display_results(graphy_results)
```

该图模型的预测指标如下所示：

measure	score
accuracy	0.978351
recall	0.924226
precision	0.943795

准确率和召回率有较大提高，但是精确率略有降低，仍然会将 8% 的链接分错。绘制 ROC 曲线并且运行以下代码来比较基本模型和图模型：

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"],
                                basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
                                 graphy_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

结果如图 8-10 所示 [1]。

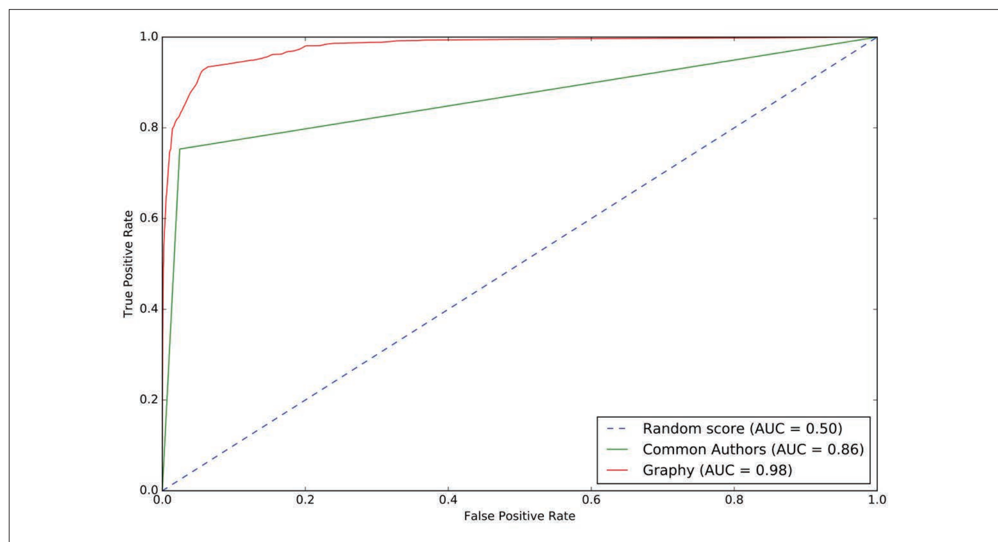


图 8-10：各模型的 ROC 曲线对比图

总体来说，似乎在朝着正确的方向前进，可视化比较有助于了解不同模型对结果的影响。

注 1：要查看图 8-10 及其他 ROC 曲线对比图的彩色版本，请访问图灵社区：http://ituring.cn/book/2694。——编者注

现在有了多个特征，还需要评估哪些特征最重要。可以依据**特征重要度**（feature importance）来对各种特征对模型预测功能的影响进行排序。这有助于评估不同算法和统计数据对结果的影响。



为了计算特征重要度，Spark 中的随机森林算法对森林中所有树的杂质减少量进行平均化处理。**杂质**（impurity）是随机分配标签错误的频率。

特征排名是与当前评估的特征分组相比较，通常要归一化。如果将某个特征排在第一位，则其特征重要度为 1.0，因为它对模型的影响程度是 100%。

以下函数创建图表来展示最有影响力的特征：

```
def plot_feature_importance(fields, feature_importances):
    df = pd.DataFrame({"Feature": fields, "Importance": feature_importances})
    df = df.sort_values("Importance", ascending=False)
    ax = df.plot(kind='bar', x='Feature', y='Importance', legend=None)
    ax.xaxis.set_label_text("")
    plt.tight_layout()
    plt.show()
```

可按如下方式调用该函数：

```
rf_model = graphy_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

运行该函数，结果如图 8-11 所示。

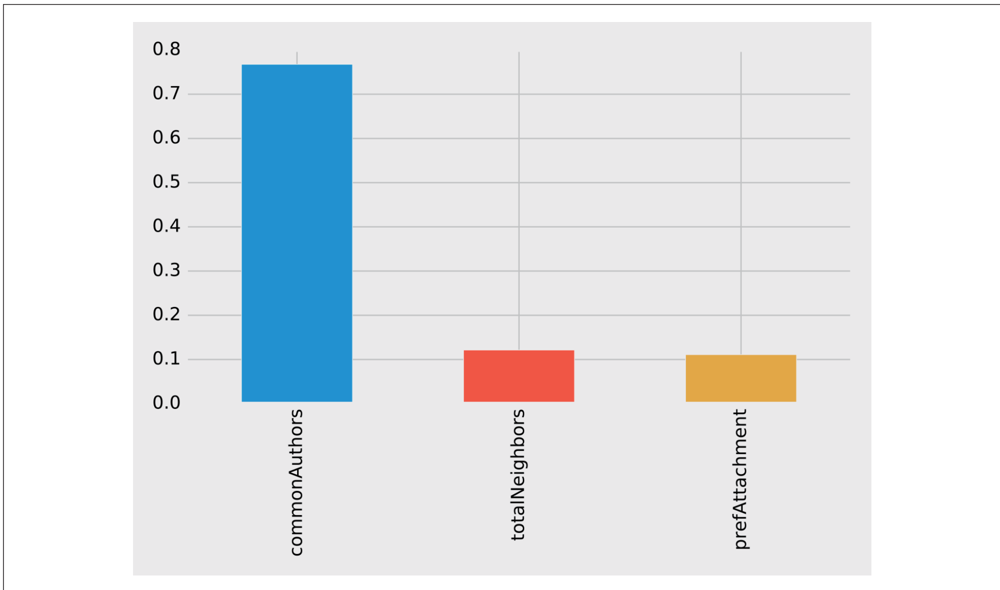


图 8-11：图模型的特征重要度

到目前为止，在用到的 3 个特征中，`commonAuthors` 最重要。

为了理解如何创建预测模型，可以使用 spark-tree-plotting 可视化随机森林中的一棵决策树。以下代码可生成一个 GraphViz 文件：

```
from spark_tree_plotting import export_graphviz

dot_string = export_graphviz(rf_model.trees[0],
    featureNames=fields, categoryNames=[], classNames=["True", "False"],
    filled=True, roundedCorners=True, roundLeaves=True)

with open("/tmp/rf.dot", "w") as file:
    file.write(dot_string)
```

然后在终端执行以下命令，将该文件可视化：

```
dot -Tpdf /tmp/rf.dot -o /tmp/rf.pdf
```

输出结果如图 8-12 所示。

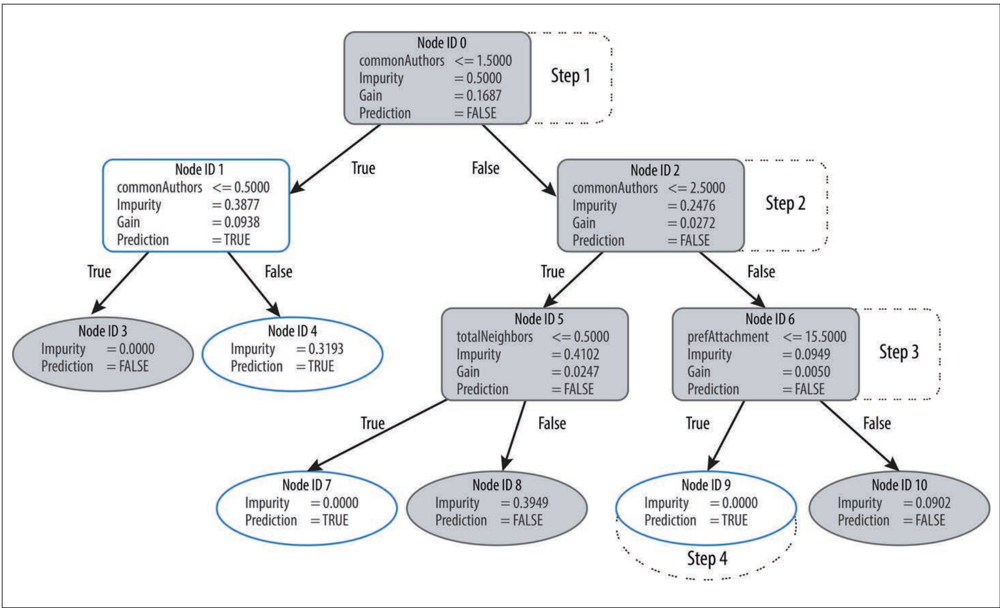


图 8-12：可视化决策树

假设要使用该决策树来预测具有某些特征的节点对是否存在链接，如下所示：

commonAuthors	prefAttachment	totalNeighbors
10	12	5

随机森林通过以下几个步骤进行预测。

1. 从节点 0 开始，由于 commonAuthors 数大于 1.5，因此沿着 False 分支到达节点 2。
2. 这里的 commonAuthors 数大于 2.5，因此沿着 False 分支到达节点 6。
3. prefAttachment 得分低于 15.5，这样就到达节点 9。
4. 节点 9 是该决策树中的一个叶节点，这意味着不需要检查其他任何条件——该节点的 Prediction 值（True）就是决策树的预测结果。
5. 最后，随机森林针对整个决策树集合求解待预测项，并基于最广泛接受的结果做出预测。

下面添加更多图特征进行研究。

8.3.8 预测链接：三角形和聚类系数

推荐解决方案通常基于某种形式的三角形度量指标进行预测，下面看看这些指标是否对本例有更多帮助。执行以下查询来计算经过节点的三角形数量及其聚类系数：

```
CALL algo.triangleCount('Author', 'CO_AUTHOR_EARLY', { write:true,
  writeProperty:'trianglesTrain', clusteringCoefficientProperty:
              'coefficientTrain'});

CALL algo.triangleCount('Author', 'CO_AUTHOR', { write:true,
  writeProperty:'trianglesTest', clusteringCoefficientProperty:
              'coefficientTest'});
```

以下函数把这些特征添加到 DataFrame 对象中。

```
def apply_triangles_features(data, triangles_prop, coefficient_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           apoc.coll.min([p1[$trianglesProp], p2[$trianglesProp]])
                                       AS minTriangles,
           apoc.coll.max([p1[$trianglesProp], p2[$trianglesProp]])
                                       AS maxTriangles,
           apoc.coll.min([p1[$coefficientProp], p2[$coefficientProp]])
                                         AS minCoefficient,
           apoc.coll.max([p1[$coefficientProp], p2[$coefficientProp]])
                                         AS maxCoefficient
    """
    params = {
        "pairs": [{"node1": row["node1"], "node2": row["node2"]}
                             for row in data.collect()],
        "trianglesProp": triangles_prop,
        "coefficientProp": coefficient_prop
    }
    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2"])
```



注意，在三角形计数和聚类系数算法中使用了前缀 min 和 max。我们需要一种方法来防止模型基于从无向图传递作者对的顺序进行学习，可以用最小计数和最大计数将这些特征按作者划分。

以下代码应用该函数训练和测试 DataFrame 对象：

```
training_data = apply_triangles_features(training_data,
                                         "trianglesTrain", "coefficientTrain")
test_data = apply_triangles_features(test_data,
                                     "trianglesTest", "coefficientTest")
```

运行如下代码展示每个三角形特征的描述性统计数据：

```
(training_data.filter(training_data["label"]==1)
 .describe()
 .select("summary", "minTriangles", "maxTriangles",
                    "minCoefficient", "maxCoefficient")
 .show())

(training_data.filter(training_data["label"]==0)
 .describe()
 .select("summary", "minTriangles", "maxTriangles", "minCoefficient",
                                                    "maxCoefficient")
 .show())
```

运行这段代码，结果如下所示：

summary	minTriangles	maxTriangles	minCoefficient	maxCoefficient
count	81096	81096	81096	81096
mean	19.478260333431983	27.73590559337082	0.5703773654487051	0.8453786164620439
stddev	65.7615282768483	74.01896188921927	0.3614610553659958	0.2939681857356519
min	0	0	0.0	0.0
max	622	785	1.0	1.0

summary	minTriangles	maxTriangles	minCoefficient	maxCoefficient
count	81096	81096	81096	81096
mean	5.754661142349808	35.651980368945445	0.49048921333297446	0.860283935358397
stddev	20.639236521699	85.82843448272624	0.3684138346533951	0.2578219623967906
min	0	0	0.0	0.0
max	617	785	1.0	1.0

请注意，本次比较中有合著关系的数据和无合著关系的数据差异不大，这可能意味着这些特征并不具有预测性。运行以下代码来训练另一个模型：

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
          "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient"]
triangle_model = train_model(fields, training_data)
```

求解模型并显示结果：

```
triangle_results = evaluate_model(triangle_model, test_data)
display_results(triangle_results)
```

三角形模型的预测指标如下所示：

measure	score
accuracy	0.992924
recall	0.965384
precision	0.958582

将每个新特征添加到之前的模型中，预测指标都有较好的增长。使用以下代码将三角形模型添加到 ROC 曲线图中：

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
                                 graphy_results["roc_auc"])

add_curve(plt, "Triangles",
          triangle_results["fpr"], triangle_results["tpr"],
                                   triangle_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

输出结果如图 8-13 所示。

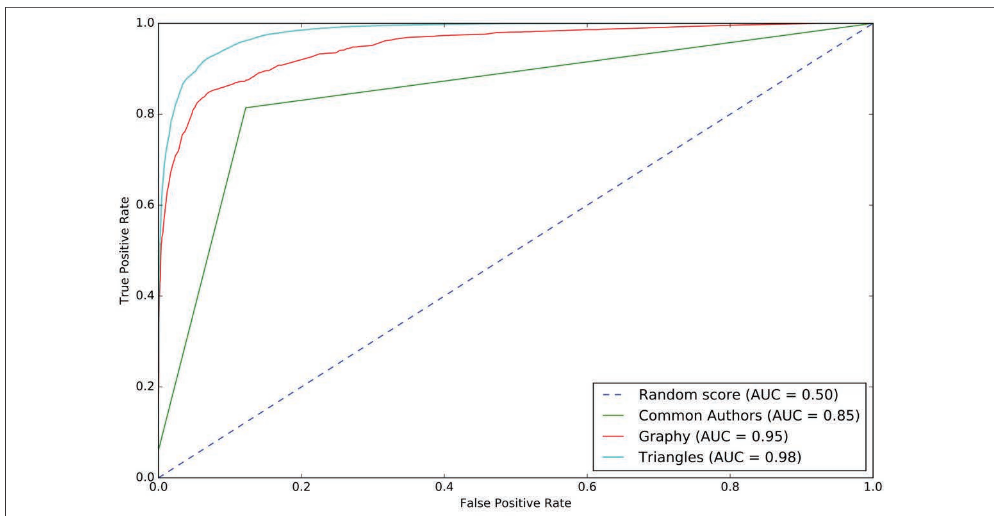


图 8-13：各模型的 ROC 曲线对比图

模型整体上得到了改进，预测指标达到了 0.98。此时情况变得更加困难，因为最容易的方法已经用过了，但是仍有改进的空间。看看特征重要度有何变化：

```
rf_model = triangle_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

运行该函数，结果如图 8-14 所示。共同作者特征对模型的影响仍然最大。也许需要探索新领域了，看看添加社团信息后会发生什么。

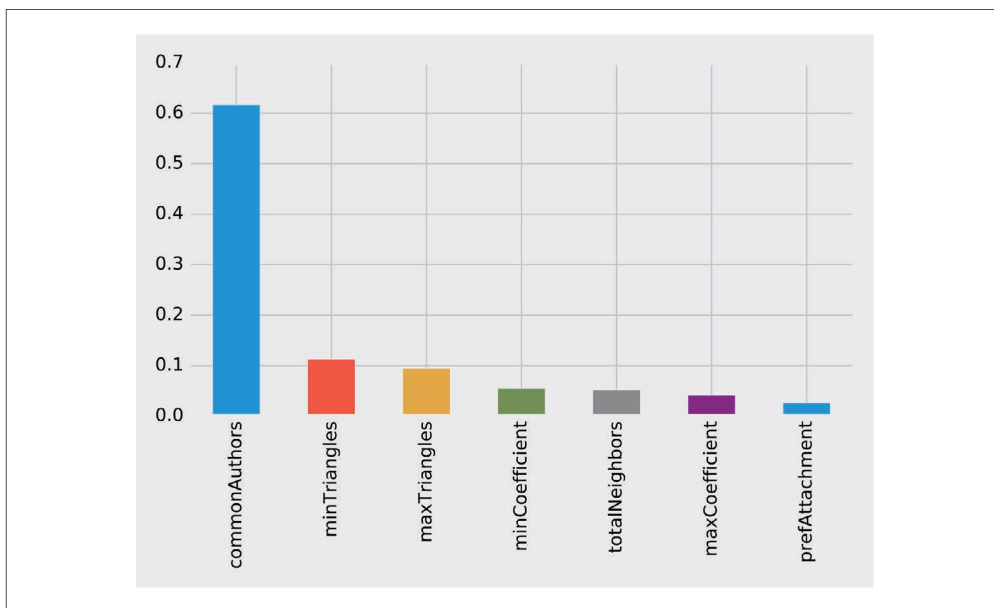


图 8-14：三角形模型的特征重要度

8.3.9 预测链接：社团发现

假设即使同一社团中的节点之间没有链接，它们之间存在链接的可能性也更大。此外，我们认为社团越紧密，存在链接的可能性越大。

首先，使用 Neo4j 中的标签传播算法计算更粗粒度的社团。执行以下查询可实现这一操作。对于训练集，该查询将把社团存储在 `partitionTrain` 性质中；对于测试集，则存储在 `partitionTest` 性质中：

```
CALL algo.labelPropagation("Author", "CO_AUTHOR_EARLY", "BOTH",
  {partitionProperty: "partitionTrain"});

CALL algo.labelPropagation("Author", "CO_AUTHOR", "BOTH",
  {partitionProperty: "partitionTest"});
```

我们使用 Louvain 模块度算法计算细粒度群组。Louvain 模块度算法将返回中间簇，对于训练集，我们将最小的簇存储在 louvainTrain 性质中；对于测试集，则将最小的簇存储在 louvainTest 性质中：

```
CALL algo.louvain.stream("Author", "CO_AUTHOR_EARLY",
                        {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTrain = smallestCommunity;

CALL algo.louvain.stream("Author", "CO_AUTHOR",
                        {includeIntermediateCommunities:true})
YIELD nodeId, community, communities
WITH algo.getNodeById(nodeId) AS node, communities[0] AS smallestCommunity
SET node.louvainTest = smallestCommunity;
```

然后创建以下函数，返回这些算法的计算结果：

```
def apply_community_features(data, partition_prop, louvain_prop):
    query = """
    UNWIND $pairs AS pair
    MATCH (p1) WHERE id(p1) = pair.node1
    MATCH (p2) WHERE id(p2) = pair.node2
    RETURN pair.node1 AS node1,
           pair.node2 AS node2,
           CASE WHEN p1[$partitionProp] = p2[$partitionProp] THEN
                     1 ELSE 0 END AS samePartition,
           CASE WHEN p1[$louvainProp] = p2[$louvainProp] THEN
                     1 ELSE 0 END AS sameLouvain
    """
    params = {
        "pairs": [{"node1": row["node1"], "node2": row["node2"]} for
                            row in data.collect()],
        "partitionProp": partition_prop,
        "louvainProp": louvain_prop
    }
    features = spark.createDataFrame(graph.run(query, params).to_data_frame())
    return data.join(features, ["node1", "node2"])
```

以下代码将该函数应用于训练和测试 DataFrame 对象：

```
training_data = apply_community_features(training_data,
                                        "partitionTrain", "louvainTrain")
test_data = apply_community_features(test_data, "partitionTest", "louvainTest")
```

运行如下代码判断节点对是否属于同一分割：

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
```

```
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('samePartition',
             F.when(F.col("samePartition") == 0, "False")
                                  .otherwise("True"))
              .groupby("samePartition")
              .agg(F.count("label").alias("count"))
              .select("samePartition", "count")
              .toPandas())
    values.set_index("samePartition", drop=True, inplace=True)
    values.plot(kind="bar", ax=axs[index], legend=None,
                title=f"Authors who {title} (label={label})")
    axs[index].xaxis.set_label_text("Same Partition")

plt.tight_layout()
plt.show()
```

运行以上代码，结果如图 8-15 所示。

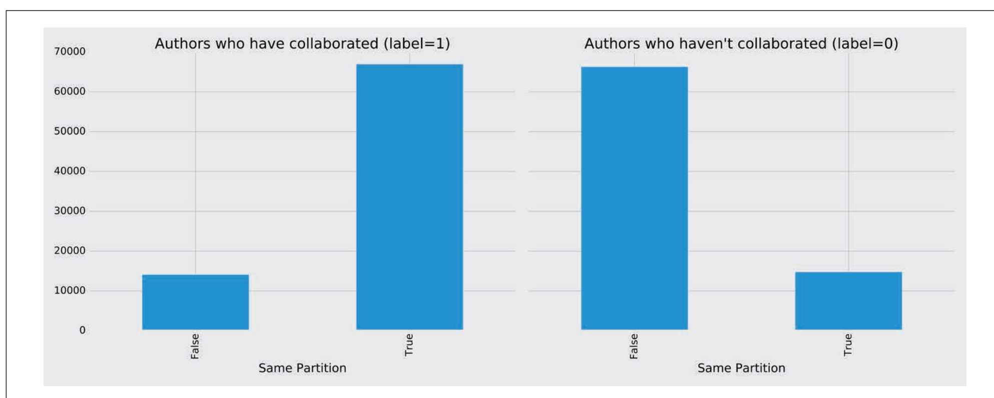


图 8-15：同一分割

看起来该特征的预测性很好——合作过的作者比未合作过的作者更有可能处于同一分割中。运行以下代码对 Louvain 簇执行相同的操作：

```
plt.style.use('fivethirtyeight')
fig, axs = plt.subplots(1, 2, figsize=(18, 7), sharey=True)
charts = [(1, "have collaborated"), (0, "haven't collaborated")]

for index, chart in enumerate(charts):
    label, title = chart
    filtered = training_data.filter(training_data["label"] == label)
    values = (filtered.withColumn('sameLouvain',
              F.when(F.col("sameLouvain") == 0, "False")
                                  .otherwise("True"))
              .groupby("sameLouvain")
              .agg(F.count("label").alias("count"))
              .select("sameLouvain", "count")
              .toPandas())
```

```
        values.set_index("sameLouvain", drop=True, inplace=True)
        values.plot(kind="bar", ax=axs[index], legend=None,
                    title=f"Authors who {title} (label={label})")
        axs[index].xaxis.set_label_text("Same Louvain")

plt.tight_layout()
plt.show()
```

结果如图 8-16 所示。

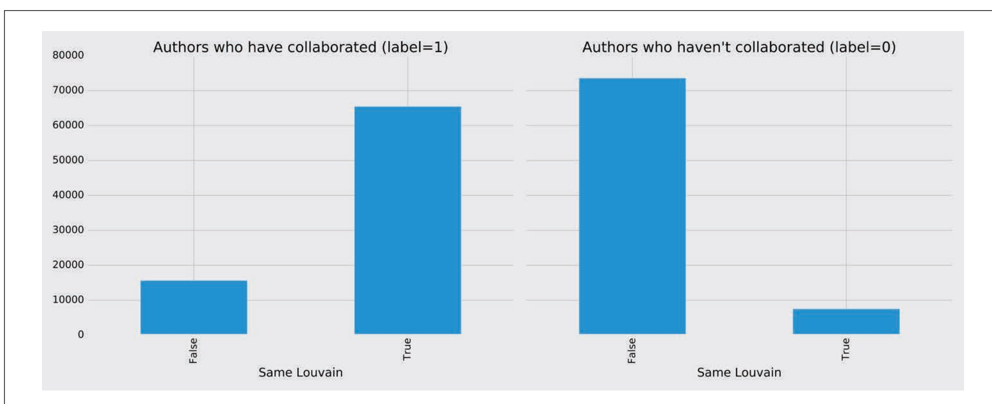


图 8-16：同一 Louvain 簇

看起来该特征的预测性很好——有合作关系的作者很可能在同一簇中，而没有合作关系的作者不太可能在同一簇中。

运行以下代码来训练另一个模型：

```
fields = ["commonAuthors", "prefAttachment", "totalNeighbors",
          "minTriangles", "maxTriangles", "minCoefficient", "maxCoefficient",
          "samePartition", "sameLouvain"]
community_model = train_model(fields, training_data)
```

然后求解模型并且展示结果：

```
community_results = evaluate_model(community_model, test_data)
display_results(community_results)
```

社团模型的预测指标如下所示：

measure	score
accuracy	0.995771
recall	0.957088
precision	0.978674

有些指标得到了改进。为了便于比较，可以运行以下代码绘制所有模型的 ROC 曲线：

```
plt, fig = create_roc_plot()

add_curve(plt, "Common Authors",
          basic_results["fpr"], basic_results["tpr"], basic_results["roc_auc"])

add_curve(plt, "Graphy",
          graphy_results["fpr"], graphy_results["tpr"],
          graphy_results["roc_auc"])

add_curve(plt, "Triangles",
          triangle_results["fpr"], triangle_results["tpr"],
          triangle_results["roc_auc"])

add_curve(plt, "Community",
          community_results["fpr"], community_results["tpr"],
          community_results["roc_auc"])

plt.legend(loc='lower right')
plt.show()
```

输出结果如图 8-17 所示。

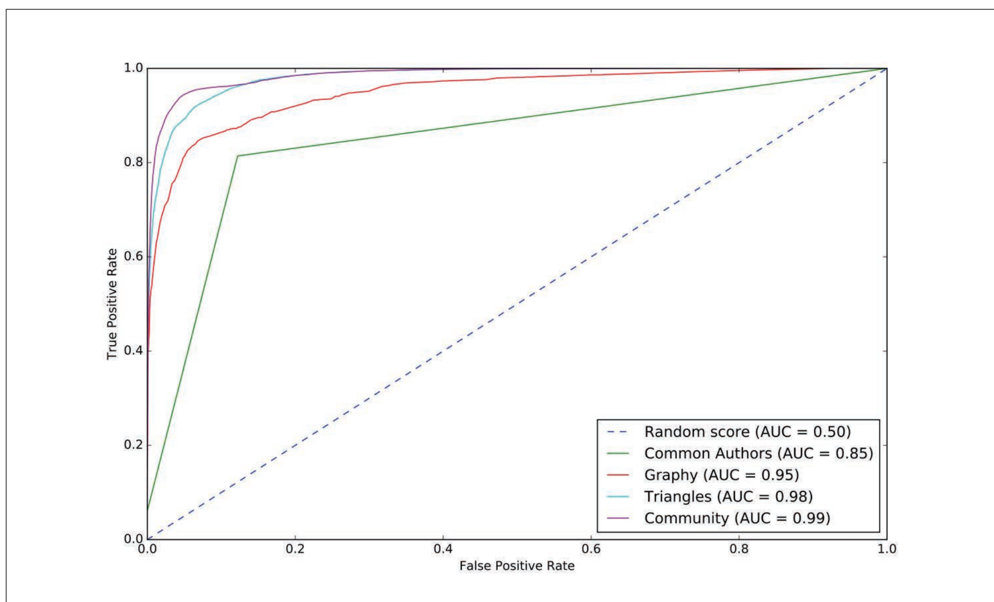


图 8-17：各模型的 ROC 曲线对比图

图 8-17 显示，添加社团模型后，指标有所改进。下面看看各个特征的重要度：

```
rf_model = community_model.stages[-1]
plot_feature_importance(fields, rf_model.featureImportances)
```

运行该函数可以得到图 8-18 所示的结果。

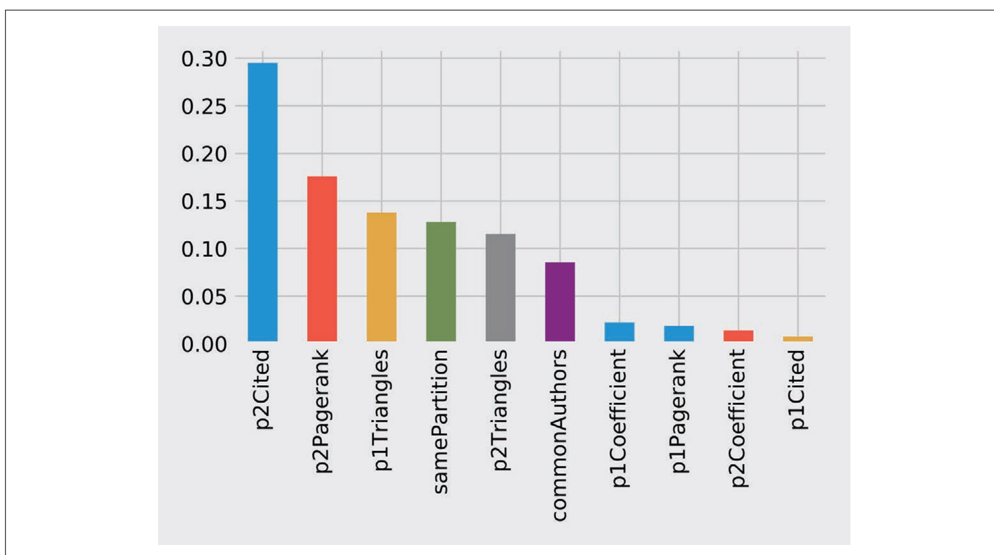


图 8-18：社团模型的特征重要度

虽然共同作者模型总体而言非常重要，但最好避免使用过于主导性的元素，以免扭曲对新数据的预测结果。社团发现算法在最后一个含有全部特征的模型中影响很大，这有助于完善预测方法。

示例表明，基于图的简单特征是良好开端，随着添加更多图特征和基于图算法的特征，预测指标持续改进。对于预测合著关系而言，现在有了一个优良且均衡的模型。

使用图进行关联特征提取可以显著地改善预测效果。图特征和图算法是否理想，这在很大程度上取决于数据的属性，包括网络域属性和图的形状属性。建议在对性能调优之前，首先探查数据中的预测性元素，并且使用不同的关联特征来验证假设。

练　　习

还有一些领域有待研究，还有构建其他模型的方法值得思考。下面这些想法值得进一步探索。

- 我们的模型对其他尚未用到的会议数据来说预测效果如何？
- 当测试新数据时，去除一些特征会发生什么？
- 如果在划分训练数据和测试数据时选取不同年份，是否会影响预测结果？
- 该数据集还包含论文之间的引用。是否可以使用该数据集生成不同的特征或预测未来的引用？

8.4 小结

本章研究了如何使用图特征和图算法来增强机器学习。首先介绍了一些基础概念，然后给出了综合使用 Neo4j 和 Spark 进行链接预测的详细示例，还演示了如何评估随机森林分类器模型，以及如何整合各种相关特征来改进预测结果。

8.5 总结

本书首先介绍了图的概念、处理平台和分析方法，然后介绍了许多关于如何在 Spark 和 Neo4j 中使用图算法的示例，最后探讨了如何用图来增强机器学习。

在现实世界中，图算法是分析各类系统背后的动力之源——它可用于防欺诈、优化呼叫路由、预测流感传播等多种场合。希望你能加入其中，动手设计独特的解决方案，充分利用如今这些高度关联的数据。

附录

额外信息及资料

本附录简单介绍一些有帮助的信息：其他算法、把数据导入 Neo4j 的另一种方法和另一个程序库，并提供一些关于查找数据集、平台帮助信息和培训等方面的资源。

其他算法

许多算法可用于图数据。本书重点介绍了经典图算法中最具代表性的算法，以及应用程序开发人员常用的算法。本书省略了像着色算法和启发式算法这样的算法，这是因为它们要么在学术案例中更受关注，要么易于推导。

如基于边的社团发现算法等其他算法也都很有意思，但是还未在 Neo4j 或 Spark 中实现。希望随着图分析应用的发展，这两种平台支持的图算法会不断增加。

还有一些算法虽然可用于图，但本质上并不是严格意义上的图算法，例如第 8 章介绍的一些在机器学习任务中使用的算法。另一个值得关注的领域是相似度算法，它常用于推荐和链接预测。相似度算法使用多种方法对节点属性等进行比较，可以计算出哪些节点彼此最为相似。

Neo4j批量数据导入和Yelp

使用 Cypher 查询语言将数据导入 Neo4j 的过程采用了事务处理方式。图 A-1 粗略地展示了这一过程。



图 A-1：基于 Cypher 的导入

这种方法适用于增量数据加载且支持多达 1000 万条记录的批量加载。在导入初始批量数据集时，Neo4j 导入工具是较好的选择。该工具直接创建存储文件，可跳过事务处理记录，如图 A-2 所示。



图 A-2：使用 Neo4j 导入工具

Neo4j 导入工具可处理 CSV 文件，并且要求这些文件具有特定标头。图 A-3 给出了该工具可以处理的 CSV 文件示例。

节点

id:ID(User)	name
1234	Bob
1235	Alice
1236	Erika

id:ID(Review)	text	stars
678	Awesome	3
679	Mediocre	2
680	Really bad	1

关系

:START_ID(User)	:END_ID(Review)
1234	678
1235	679
1236	680

图 A-3：Neo4j 导入处理的 CSV 文件格式

Yelp 数据集的规模意味着采用 Neo4j 导入工具是最佳选择。该数据是 JSON 格式的，因此首先需要将其转换为 Neo4j 导入工具所需的格式。图 A-4 展示了需要进行转换的 JSON 示例。

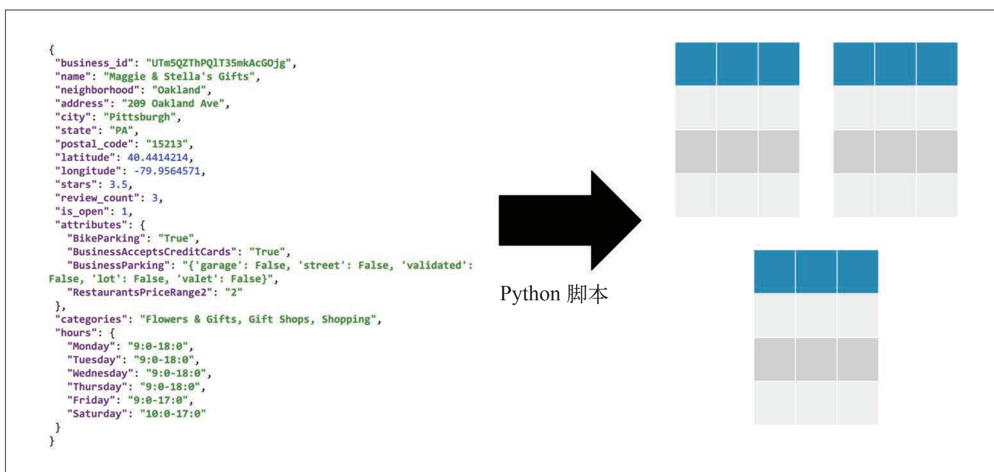


图 A-4：将 JSON 转换为 CSV

使用 Python 可以创建简单脚本，将数据转换为 CSV 文件，然后将其导入 Neo4j 中。关于该操作的详细说明，参见 Neo4j 官网。

APOC和其他Neo4j工具

APOC（Awesome Procedures on Cypher）包含数百个程序和函数，用于辅助完成集成、清理、转换数据等常见任务并提供常规辅助函数。APOC 是面向 Neo4j 的标准库。

Neo4j 还提供了其他一些可与其图算法库一起使用的工具，比如支持无代码开发的算法“游乐场”应用程序。这些都可以在其图算法开发网站上找到。

查找数据集

查找符合测试目标或假设的图数据集富有挑战性。除了阅读研究论文，还要考虑探查网络数据集的索引。

- SNAP（Stanford Network Analysis Project）包含几个数据集以及相关论文和使用指南。
- ICON（ Index of Complex Networks）是一个可搜索的索引，包含来自网络科学多个领域的高质量网络数据集。
- KONECT（Koblenz Network Collection）包含用于网络科学研究的多种大规模网络数据集。

大部分数据集需要经过一定的处理才能转换为更易用的格式。

Spark和Neo4j平台帮助信息

Spark 和 Neo4j 有很多在线资源。针对具体问题，可以分别访问以下社区。

- 对于 Spark 常见问题，可订阅 Spark 社区网页上的 users@spark.apache.org。
- 对于 GraphFrames 相关问题，可使用 GitHub 问题追踪器。
- 所有关于 Neo4j 的问题（包括图算法问题），都可访问 Neo4j 在线社区。

培训资源

还有许多优质资源有助于学习图分析。搜索关于图算法、网络科学和网络分析的课程或图书，可以找到许多资源。以下列举比较好的在线学习资源：

- Coursera 上的 Python 实用社交网络分析课程；
- Leonid Zhukov 的社交网络分析 YouTube 系列课程；
- 斯坦福大学的网络分析课程，包括视频讲座、阅读清单和其他资源；
- Complexity Explorer 提供的复杂性科学在线课程。

关于作者

马克·尼达姆（**Mark Needham**）是图的倡导者和开发者关系工程师。他致力于帮助用户运用图和Neo4j，善于针对富有挑战性的数据问题构建复杂的解决方案。马克在图数据方面拥有丰富的专业知识，此前协助构建了Neo4j的因果聚类系统。他通过广受欢迎的个人博客记录自己从事图相关研究的经历。他的Twitter账号是@markhneedham。

埃米·E. 霍德勒（**Amy E. Hodler**）是Neo4j的人工智能和图分析项目经理。她热爱网络科学，提倡使用图分析来揭示现实网络的结构并预测动态行为。埃米帮助团队运用新方法为EDS、微软、惠普、日立和Cray Inc.等公司创造新的商机。埃米热爱科学和艺术，对复杂性研究和图论有着浓厚的兴趣。她的Twitter账号是@amyhodler。

关于封面

本书封面上的动物是欧洲花园蜘蛛（也称十字园蛛），常见于欧洲和北美洲，当年随欧洲移民登陆北美洲。

欧洲花园蜘蛛体长不到3厘米，体色为斑驳的棕色，带有苍白斑纹，背部的几处斑纹形同一个小十字架，故得名十字园蛛。这些蜘蛛在当地很常见，多在夏末出现，因为此时它们发育成熟并开始结网。

欧洲花园蜘蛛属金蛛科，它们会结一张圆形的网来捕捉小昆虫等猎物。为了保证捕食效率，它们通常在晚上使用和修补蛛网。这种蜘蛛通常保持隐蔽，一条腿停在与网相连的“信号线”上，这条“信号线”的颤动会提醒蜘蛛有猎物上门，然后蜘蛛迅速行动，咬住猎物并将其杀死，之后注入特殊的酶以供享用。当它们的网受到捕食者破坏或无意间干扰时，欧洲花园蜘蛛会用腿晃动蛛网，然后通过一根蛛丝落到地上。待危险过去后，蜘蛛会用蛛丝重新结网。

这种蜘蛛的寿命只有一年：春季孵化，夏季成熟，年末交配。雄蛛在接触雌蛛时很小心，这是因为有时雌蛛会杀死并吃掉雄蛛。交配之后，雌蛛会为自己的卵织一个厚厚的茧，然后在秋天死去。

由于它们很常见，可以很好地适应已被人类干扰的栖息地，因此人们对它们进行了深入研究。1973年，名为Arabella和Anita的两只雌性蜘蛛参加了美国国家航空航天局在太空实验轨道飞行器上的实验，实验目的是测试零重力对蛛网结构的影响。花了一段时间适应失重环境后，Arabella先结了一部分网，而后织成了一张完整的圆形网。

O’Reilly图书封面上的许多动物濒临灭绝，它们是自然界所剩无几的瑰宝。

封面图片是Karen Montgomery提供的彩色插图，基于*Meyers Kleines Lexicon*上的一幅黑白版画创作。



微信连接



回复“算法”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

数据分析之图算法：基于Spark和Neo4j

莎士比亚曾说，世界是一个大舞台。在今天看来，世界是一张大图！将人物和事件视作节点，将节点之间的关系连成线，我们就能将错综复杂的关系网络转化为图，通过图分析洞悉复杂问题的本质。图算法已经广泛应用于数据分析领域，营销归因分析、欺诈网络检测、客户旅程建模、安全事故原因分析，甚至连莎士比亚戏剧的剧情分析，都会用到图算法。

学习图算法有助于利用数据间的关系研究智能解决方案，并构建增强机器学习模型。本书作者来自Neo4j公司，在图分析领域深耕多年。你将跟随他们领略美妙的图算法世界，并利用流行平台Spark和Neo4j实现常用的图算法。

- **了解如何利用图分析揭示数据的预测性特征**
- **了解如何实现近20种流行的图算法**
- **了解各种图算法的适用场景**
- **跟随示例在Spark和Neo4j中应用图算法**
- **结合Spark和Neo4j创建机器学习工作流程**

马克·尼达姆（Mark Needham），Neo4j公司开发者关系工程师，Neo4j认证专家，曾深度参与Neo4j因果集群的开发工作。马克致力于帮助客户运用图数据库，善于针对富有挑战性的数据问题构建综合的解决方案。

埃米·E.霍德勒（Amy E. Hodler），Neo4j公司图分析与人工智能项目总监，热爱网络科学，在图分析项目的开发和运营方面有着丰富的经验，曾成功带领团队为EDS、微软、惠普等公司创造新的商机。

“从基本概念到基础算法，再到处理平台和实用案例，作者为精彩的图算法世界编写了一本内容翔实的指南。”

——Kirk Borne博士
博思艾伦咨询公司
首席数据科学家兼执行顾问

“这本实用指南介绍了如何使用图算法检测模式和结构，从而洞察存在连接关系的数据。我强烈推荐图数据库开发人员阅读。”

——Luanne Misquitta
GraphAware工程副总裁

PROGRAMMING / ALGORITHMS

封面设计：Karen Montgomery　张健

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议　计算机/编程与开发/算法

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



扫码领取
随书代码资料


ISBN 978-7-115-54667-8

定价：79.00元